Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Escalonador de Tarefas para o Plataforma de Nuvens Federadas BioNimbuZ Usando Beam Search Iterativo Multiobjetivo

Willian de Oliveira Barreiros Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientadora
Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo

Brasília
2016

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Escalonador de Tarefas para o Plataforma de Nuvens Federadas BioNimbuZ Usando Beam Search Iterativo Multiobjetivo

Willian de Oliveira Barreiros Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo (Orientadora)
CIC/UnB

Prof.ª Dr.ª Alba Cristina M. A. de Melo
CIC/UnB

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof. Dr. Ricardo Zelenovsky
Coordenador do Curso de Engenharia da Computação

Brasília, 25 de Janeiro de 2016

Dedicatória

Ao meu pai, que sempre foi meu guia e mais verdadeiro amigo, à minha mãe, que como meu *rubber duck* sempre me ajudou nos trabalhos, mesmo sem entender sobre o que se tratavam, e a meu irmão e minhas primas, que tiveram que suportar minha barulhenta rotina noturna de trabalho.

Agradecimentos

Agradeço primeiramente a minha orientadora, Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo, pela paciência e empenho dedicado a este trabalho, que tornou possivel a execução deste. Aos meus colegas do grupo BioNimbuZ, pelo companherismo e camaradagem em todos os momentos. À Prof.ª Dr.ª Genaína Nunes Rodrigues pelo auxílio acadêmico a mim dispensado. Ao Prof. Dr. Rodrigo Bonifacio de Almeida pelas oportunidades e conhecimentos práticos adiquiridos sob sua supervisão. Finalmente, ao Prof. Dr. Alexandre Zaghetto, pela confiança e oportunidade oferecida desde o início do curso, e que através de suas aulas tive a certeza de ter escolhido o curso certo.

Resumo

O problema de escalonamento de tarefas em sua formulação genérica é NP-Completo, caracterizando-o assim como um problema computacionalmente difícil. Ao levar esse problema para o ambiente distribuído de federação de nuvens computacionais o mesmo se torna ainda mais complicado. Dessa forma, uma política de escalonamento eficiente é altamente desejável para qualquer plataforma de federação de nuvens. Neste contexto, este trabalho visa a proposta e implementação de uma nova política de escalonamento para a plataforma de nuvem federada BioNimbuZ, baseada no algoritmo de busca combinacional *beam search*. O algoritmo proposto foi concebido com o objetivo de convergir rápidamente para boas soluções. Além disso, ele deve ser multiobjetivo, pois visa minimizar ambos tempo e custo de execução via o uso de uma frente de pareto. Os resultados dos testes iniciais mostraram um bom desempenho do algoritmo em relação à velocidade de convergência e ao tempo de execução total do algoritmo.

Palavras-chave: Computação em Nuvem, Escalonamento de Tarefas, *Beam Search* Iterativo, BioNimbuZ.

Abstract

Task scheduling, on its general formulation, is NP-Complete, making it thus a computationally hard problem. Even more so in the the distributed computing environment of cloud federations. Therefore, an efficient scheduling policy is a crucial component for any cloud federation platform. On this context, this work defines a new scheduling policy for the BioNimbuZ cloud federation platform, based on the beam search algorithm. The proposed algorithm was conceived with the goal of having a fast convergence to good solutions. Moreover, the algorithm should be multiobjective, by minimizing both time and cost of execution, through the use of a pareto frontier. Preliminary tests showed that the algorithm had good performance when analyzing the speed of convergence and the overall execution time of the algorithm.

Keywords: Cloud Computing, Task Scheduling, Iterative Beam Search, BioNimbuZ.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em computação, a busca por cada vez mais desempenho computacional, atrelado à confiabilidade e ao custo começa nos anos 1960, nos quais os conceitos de computação como serviço de utilidade foram originados. Uma parte da comunidade científica faz menção a Licklider e Taylor [27], que propuseram o conceito de *Galactic Network*, uma rede global interconectada na qual seria possível acessar programas e dados de qualquer lugar, uma ideia necessária ao conceito de nuvem. No mesmo período John McCarthy ministrou uma importante palestra no MIT em 1961 [22], onde previa que o futuro da computação seria como um serviço de utilidade.

Visando suprir a demanda por poder computacional, os supercomputadores apareceram em 1964 com o *Control Data Corporation* 6600 [8], enquanto *clusters* começaram com o ARCnet em 1977 [1]. Essas duas propostas são bastante diferentes, no sentido de que *clusters* são montados pela combinação de computadores normalmente homogéneos, usando hardware e software discreto, enquanto supercomputadores possuem hardware e software proprietários, adaptados para computação de alto desempenho. Outra diferença é o custo, sendo que supercomputadores são mais caros. De qualquer forma, ambos agem como um único sistema, estando, normalmente, na mesma localidade, geograficamente [45].

Nos anos 1990 houve uma popularização dos clusters. Isto se deve principalmente ao fato que a montagem de um *cluster* era simplificada e mais barata quando comparada com o custo de um supercomputador. Além disso, em 1997 *clusters* começaram a entrar para a lista Top 500, que contemplava as 500 máquinas mais rápidas do mundo. Outra razão que encorajava a sua adesão era a forma de uso na qual ela foi concebida, como um serviço de utilidade confiável e facil de usar, orientada a aplicações cientificas de uso intensivo de memória, que necessitavam compartilhamento de recursos.

Mesmo com *grids* computacionais tendo uma grande similaridade ao modelo de nuvens, no sentido de combinação horizontal de recursos, o primeiro serviço de computação como

utilidade chegou em 1999 da SalesForce. Este serviço tinha como motivação a criação de um Software como Serviço (SaaS) para otimizar o *Customer Relationship Management* [13].

A Amazon deu o próximo passo em 2002, disponibilizando o *Amazon Web Services* (AWS) [2]. O serviço prestado seria o de *Plataform as a Service* (PaaS), disponibilizando *hosting* para aplicações, que começou como uma demanda interna por computação abstraída como plataforma. Assim, quatro anos depois, a Amazon acrescentou ao seu portfólio do AWS, o serviço de instâncias de hardware sob demanda, ou *Infrastructure as a Service* (IaaS) [2].

Assim como *clusters* foram uma maneira de combinar os recursos disponíveis na época de sua idealização para aumentar a capacidade dos mesmos em um conjunto, tem-se agora federações de nuvens, que tem por objetivo a combinação de várias provedoras de nuvens para o balanceamento de carga e melhoria na disponibilidade e na confiabilidade do ambiente.

Neste contexto, o uso de algoritmos que otimizem a execução de tarefas pela definição do local de execução é tão importante como o próprio ambiente subjacente na qual tais tarefas são executadas. Dessa forma, este trabalho propõe a implementação de um algoritmo de escalonamento que aproveite todas as vantagens de um ambiente de federação de nuvens.

1.1 Objetivos

Este trabalho tem como objetivo a proposta de um algoritmo de alocação de tarefas no ambiente de nuvens federadas. Este algoritmo deve ser demonstrado melhor que a abordagem atualmente usada no ambiente BioNimbuZ (ACOSched), que será a plataforma na qual o escalonador será integrado. Para cumprir o objetivo geral, faz-se necessário atingir os seguintes objetivos:

- Implementar um algoritmo de busca combinacional;
- Definir regras multiobjetivo;
- Implementar um algoritmo de otimização da busca;
- Tornar o algoritmo incremental;
- Medir a sua performance com testes de performance e reatividade a diferentes instâncias.

1.2 Organização

Em adição a este capítulo introdutório, este trabalho está dividido em mais quatro capítulos. No segundo capítulo serão feitas as definições de computação em nuvem e federação de nuvens de uma forma mais orientada ao meio acadêmico. Também será definida a plataforma BioNimbuZ, a qual é usada como base para o escalonador proposto neste trabalho. Em seguida, será feita uma analise detalhada do problema de escalonamento em um escopo geral, seguida, por fim, de uma análise da literatura sobre algoritmos de escalonamento para o ambiente de nuvens.

No Capítulo 3 é feita a descrição do algoritmo de escalonamento proposto neste trabalho. Esse é composto pela definição de objetivos gerais, pela definição do algoritmo de pareto usado, pela descrição do algoritmo de ordenação de busca, pela descrição de cada estágio do algoritmo e pelo algoritmo de poda proposto. Também são definidas a estrutura de busca utilizada e as variáveis internas de cada nó.

O Capítulo 4 tem como objetivo demonstrar a eficiência do algoritmo proposto pela definição de métricas a serem observadas e pelos dados do teste. No primeiro momento é feita a descrição do ambiente e dos próprios testes a serem executados. Finalmente, é feita uma análise de cada uma das métricas propostas, indicando conclusões sobre o algoritmo proposto.

Finalizando, o último capítulo relata os objetivos alcançados, vantagens, e desvantagens do algoritmo proposto neste trabalho. Em seguida, são apresentados alguns possíveis trabalhos futuros que possam ser realizados a partir deste para melhorar ainda mais a eficiência do mesmo.

Capítulo 2

Computação em Nuvem

Este capítulo introduz conceitos básicos sobre computação em nuvens, federação de nuvens e encerra com um detalhamento do conceito de escalonamento. Este conceito é expandido em duas classes de algoritmos que serão detalhadas, algoritmos genéticos e combinacionais, as quais serão explanadas no foco de aplicações em nuvens. Por fim, serão apresentados os trabalhos relacionados usados como referência para o algoritmo proposto neste trabalho.

2.1 Definição de Computação em Nuvem

Computação em nuvem vem sendo usada como uma alternativa aos *datacenters* tradicionais, oferecendo um melhor custo-beneficio, extensibilidade, além de outras vantagens [16]. A sua definição é bem consolidada, havendo pelo menos duas definições de grandes institutos, o NIST (*National Institute of Standards and Technology*) [32] e o ETSI (*European Telecommunications Standards Institute*) [18].

De um modo geral, computação em nuvem pode ser definida como um modelo que disponibiliza recursos computacionais configuráveis de maneira ubíqua, sob demanda, pela *internet*, nos quais esses recursos podem ser reservados e liberados com mínimo esforço de gestão ou interação do provedor deste serviço [15].

A definição de computação em nuvem pode ser, então, dividida em três, as quais são caracteristicas essenciais, modelos de serviço e modelo de implantação [32]. Em relação às caracteristicas essenciais de um ambiente de nuvem, destacam-se:

- *Self-service* sob demanda: O cliente deve poder alocar e desalocar dinamicamente recursos sem a necessidade de intervenção direta do provedor;
- Acesso amplo via rede: A nuvem deve estar disponível amplamente na *internet*;
- Combinação de recursos: A provedora deve alcançar uma alta capacidade computacional pela combinação/união de hardwares/softwares;

- Elasticidade: A provedora deve poder ajustar a sua capacidade computacional de acordo com a demanda;
- Medição de serviço: Deve ser possível o monitoramento de cada instância virtualizada, a fim de gerar métricas de uso.

Por outro lado, o modelo de serviço define qual o nível de abstração de recursos que uma provedora usa. Existem três níveis basicos: *Software as a Service* (SaaS), *Plataform as a Service* (PaaS), e *Infastructure as a Service* (IaaS) [18, 32]. SaaS é o nivel mais alto, disponibilizando para o cliente um software que executa na infraestrutura da nuvem, abstraindo essa infraestrutura para o cliente. Por outro lado, ao usar uma instância de PaaS, é provido ao cliente uma plataforma onde ele poderá executar aplicações específicas na mesma (e.g. aplicações web), novamente, sem acesso direto à infraestrutura subjacente. Finalmente, caso o cliente necessite de um maior controle sobre a instância alocada, existe o modelo IaaS. Nesse caso, o cliente tem controle em nível de máquina virtual. Vale ressaltar que este trabalho usa apenas instâncias IaaS.

Finalmente, em relação ao modelo de implementação, uma nuvem pode ser Privada, Comunitária, Pública ou Híbrida [32]. Uma nuvem Privada é, por definição, um ambiente computacional controlado por uma organização, cujos recursos são utilizados por clientes da mesma. Uma nuvem Comunitária consiste no mesmo conjunto de recursos computacionais, usados por uma comunidade de organizações com interesses em comum. Uma nuvem é dita Pública caso seus recursos estejam abertos para uso geral. Uma combinação de dois ou mais modelos de organização descritos é definida como uma nuvem Híbrida.

2.2 Federação de Nuvens

O conceito de federação de nuvens também está bem estabelecido, todavia, não existe um padrão de implementação consolidado [26]. Por definição, uma federação de nuvens é um sistema cooperativo de duas ou mais provedoras de serviço de computação em nuvem [20]. Esta cooperação, em nível de cliente de nuvens, é interessante por quebrar a dependência por um único provedor, melhorando assim a disponibilidade de serviço da federação, reduzindo custos, e também aumentando o número de combinações de instâncias.

Existem basicamente duas maneiras na qual distintos ambientes de nuvem podem cooperar, verticalmente ou horizontalmente [19]. Caso haja uma ordem de precedência na escolha de uma nuvem pela federação, ou haja uma pilha de nuvens, na qual uma nuvem de um nível mais alto use recursos de um nível inferior, esta é dita uma federação vertical [26]. No caso de federações horizontais não existem tais restrições de acesso. Vale ressaltar que ambos os modelos podem ser usados conjuntamente. Um exemplo são as

plataformas *Eucalyptus* [9] e *OpenNebula* [11], que similarmente tem suporte ao modelo de federação vertical, o qual é construido a partir de uma federação horizontal de nuvens [19].

Além da *Eucalyptus* [9] e *OpenNebula* [11], existem diversas outras propostas de arquiteturas para federação de nuvens. Assim, este trabalho tem seu foco na arquitetura proposta pela plataforma BioNimbuZ [43].

2.3 BioNimbuZ

O BioNimbuZ é uma plataforma de nuvens federadas horizontal com o objetivo de executar um grande conjunto de *pipelines* de tarefas de Bioinformática [43]. Ele foi desenvolvido originalmente em 2013 e está em contínuo desenvolvimento pelo laboratório LABID (Laboratório de Bioinformática e de Dados) da Universidade de Brasília [43]. Internamente, o BioNimbuZ é dividido logicamente em três camadas, as quais são Aplicação, Núcleo e Infraestrutura, conforme apresentado na Figura 2.1. Para um melhor entendimento cada uma dessas camadas será descrita em detalhes nas próximas seções.

2.3.1 Camada de Aplicação

A camada de aplicação provê a abstração de execução de tarefas de Bioinformática, recebendo *pipelines* descrevendo as tarefas a serem executadas. Essas tarefas são descritas pelo serviço de Bioinformática que deve ser executado e parâmetros de entrada para o mesmo. Estão inclusos nos parâmetros quaisquer arquivos a serem usados como entrada, podendo ser fornecidos diretamente pelo usuário ou pela saída de outra tarefa. Os *pipelines* são enviados, em sua totalidade, diretamente ao núcleo do BioNimbuZ via RPC (*Remote Procedure Call*). Esta camada possui dois tipos de interface com o usuário, uma interface via linha de comando e uma interface *web*.

2.3.2 Camada de Infraestrutura

Nesta camada estão presentes dois elementos críticos ao funcionamento do BioNimbuZ, os serviços de comunicação e os *plugins* de instâncias. Os serviços usados para realizar a comunicação entre os nós do BioNimbuZ são o Zookeeper [5] e o Avro [4].

O Avro, sendo um sistema de RPC criado pela Apache [3] para simplificar o processo de serialização de objetos. O ZooKeeper funciona como um banco de dados centralizado, para um ambiente distribuído, usado para sincronizar serviços e grupos distribuídos. Para o BioNimbuZ, o ZooKeeper é usado para armazenar descritores de nós, de tarefas e *pipelines* que estão sendo executados e que serão executados futuramente, e de arquivos

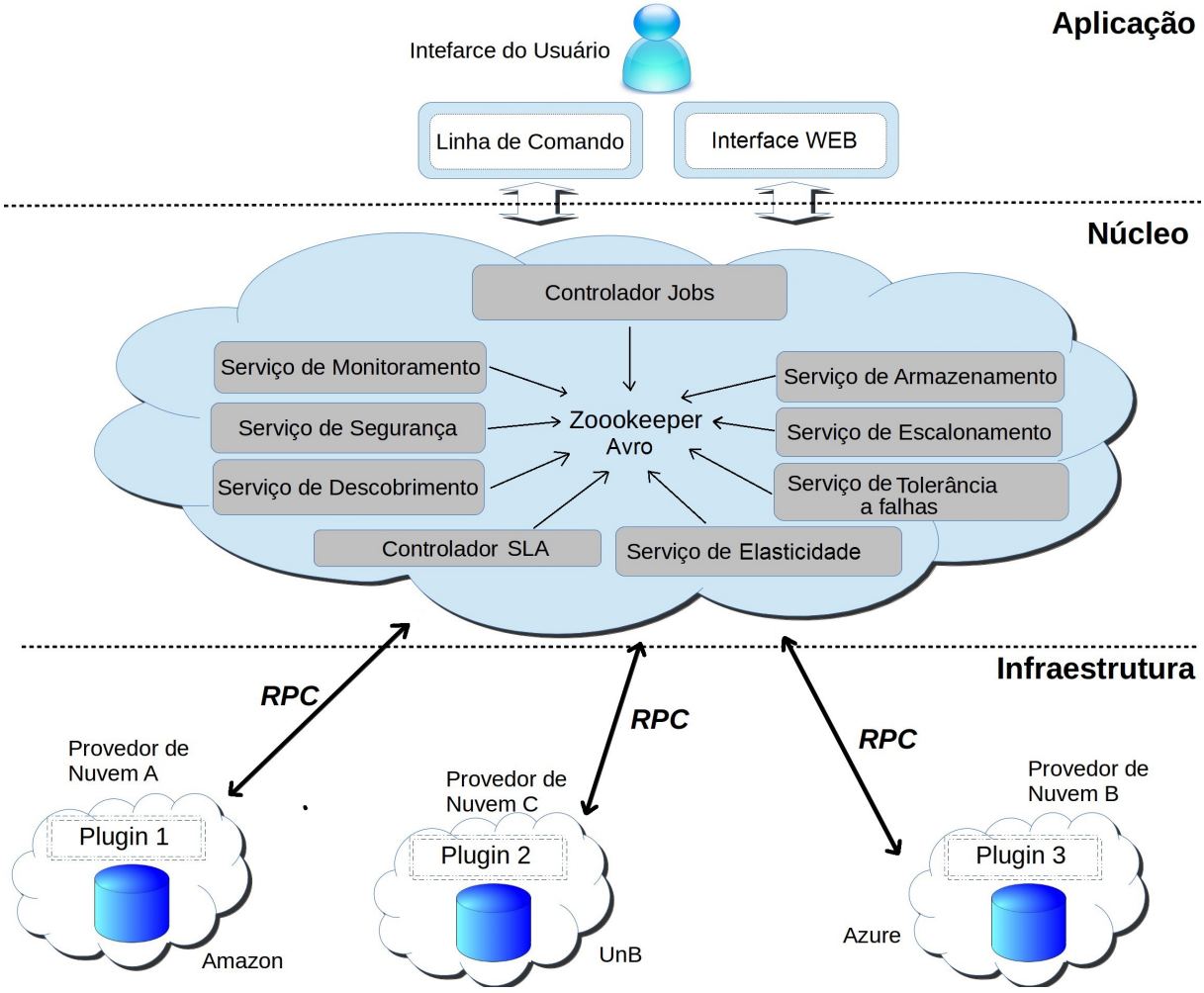


Figura 2.1: Arquitetura do BioNimbuZ.

armazenados na federação. Além disso, o ZooKeeper possui um sistema de *triggers*, que são disparados em casos de alteração na estrutura ou no conteúdo dos dados armazenados. Por fim, um dos motivos que determinaram a escolha do ZooKeeper foi a necessidade de um sistema que garantisse consistência de dados em vista a escalabilidade no número de nós suportados.

O sistema de plugins foi uma solução encontrada para suportar a integração de qualquer plataforma de nuvem de maneira totalmente transparente para o usuário e para os provedores.

2.3.3 Camada de Núcleo

A camada de núcleo, por sua vez, é formada por um conjunto de serviços independentes, que se comunicam diretamente via Avro, e indiretamente (e, principalmente) via ZooKeeper. Esses serviços são o Controlador de *Jobs*, o Controlador de SLA (*Service*

Level Agreement), os Serviços de Monitoramento, de Descoberta, de Armazenamento, de Segurança, de Tolerância a falhas, de Elasticidade e de Escalonamento.

Do ponto de vista lógico, o BioNimbuZ executa uma versão reduzida de si em cada nó de processamento (i.e., recurso/instância na nuvem), contendo os serviços necessários para receber novas tarefas e executá-las. Para cada nó são garantidos os requisitos mínimos da política de armazenamento, como replicação, e a comunicação com o nó raiz, ou mestre, no qual são realizadas todas as decisões do BioNimbuZ [42].

Os serviços de monitoramento e de descoberta são responsáveis pela descoberta de novos nós e pelo monitoramento de atividades dos mesmos, respectivamente. Entre as atividades de monitoramento tem-se o monitoramento do estado de cada nó, o monitoramento de tarefas e de *pipelines* (por exemplo, o término da execução de uma tarefa ou a criação de um novo arquivo), e a atualização do estado desses junto ao ZooKeeper. Esse estado no ZooKeeper, por sua vez, é acessível a todos os demais serviços.

O serviço de armazenamento é responsavel pela decisão de quantas réplicas de um arquivo devem ser salvas e a localização de cada uma delas. Esse serviço funciona de acordo com o padrão *Strategy* [21], na qual existe uma política de armazenamento que é chamada pelo serviço de armazenamento no evento de *upload* de arquivos [34].

O serviço de controle de SLA monta os contratos que são firmados com os clientes do BioNimbuZ, também garantindo que tais contratos não sejam violados, e que, caso isto ocorra, que haja uma auditoria sobre os motivos.

O serviço de elasticidade é responsável pelo aumento/redução dinamicamente do número de VMs instanciadas nos provedores de IaaS. Esse número de instância é alterado nos casos de necessidade do sistema ou de acordo com o estado atual do mesmo (i.e., número de tarefas sendo executadas, quantidade de recursos ociosos, número de tarefas a serem escalonadas/executadas, etc).

Os serviços de controle de *jobs* e tolerância a falhas trabalham juntos, garantindo que todas as tarefas a serem executadas sejam enviadas ao nó de processamento correto, e que seja executada com sucesso. Caso ocorra alguma falha relacionada à infraestrutura de execução de tarefas, a mesma é reiniciada em outro nó. No caso de falha causada pelo usuário a tarefa é finalizada.

O serviço de segurança ainda está em um estágio inicial, no qual apenas um sistema de integridade de arquivos e um sistema de controle de acesso foram implementados [17]. Este serviço codifica cada arquivo a ser trocado entre nós do BioNimbuZ, garantindo a transmissão sem adulteração ou leitura não autorizada dos arquivos.

Por fim, o serviço de escalonamento possui uma arquitetura semelhante ao serviço de armazenamento, permitindo a escolha de uma política de escalonamento entre um conjunto de implementações já existentes. O serviço de escalonamento recebe via ZooKeeper

pipelines, coleta as tarefas executáveis dos mesmos (i.e. sem dependências), e chama o escalonador periodicamente. Atualmente, existem duas políticas já implementadas, o ACOSched e o DynamicAHP, sendo a primeira a usada atualmente [42].

O algoritmo de escalonamento proposto neste trabalho será integrado ao BioNimbuZ como uma nova política de escalonamento. Dessa forma, ele receberá uma lista de tarefas independentes e uma lista de recursos a serem usados, e retornará o mapeamento das tarefa para os recursos, denotando assim um algoritmo de alocação de tarefas. Isso significa que as tarefas não serão escalonadas de maneira cronologica *a priori*, e sim escalonadas *on the fly*, assim que for possivel a sua execução. Porém, para a definição de tal algoritmo, é necessário antes a definição do ambiente no qual ele escalonará tarefas, e o estudo de abordagens já utilizadas para tal.

2.4 Escalonamento no Ambiente de Nuvens

A criação de algoritmos de escalonamento é um problema clássico da área da Ciência da Computação, sendo um dos mais complexos. A definição usada neste trabalho para escalonamento é a associação de um conjunto definido de tarefas independentes a um conjunto de recursos computacionais com o objetivo de executá-las. Embora existam vários algoritmos que tratem esse problema [37], poucos são aplicáveis ao domínio deste trabalho.

Primeiramente, é interessante definir o domínio do problema de escalonamento abordado neste trabalho. Assim sendo, sabe-se que as tarefas a serem escalonadas podem possuir oportunidades de paralelismo quando existe um volume de tarefas suficientemente grande. Além disso, o ambiente usado para executar tais tarefas pode ser um conjunto de recursos heterogêneos. Por fim, esse problema de escalonamento é multi-modal em geral, ou seja, pode possuir mais de uma solução boa localmente. Essas caracteristicas tornam o problema de escalonamento de tarefas, para o ambiente de nuvem, mais difícil [37]. Entre as abordagens pesquisadas para resolver esse problema dois modelos se destacam, algoritmos genéticos [25, 28, 46] e algoritmos combinacionais [29, 30, 40].

Algortimos genéticos são algoritmos baseados no processo evolutivo, em que existe uma população de soluções na qual cada indivíduo (solução) é definido por um conjunto de características genéticas. Para cada conjunto de características é extraído um valor de *fitness*, indicando a aptidão de cada indivíduo. Indivíduos podem, então, cruzar e produzir prole com a combinação de características de seus pais. Existe também o processo de mutação natural, onde um indivíduo muda suas características entre gerações de maneira aleatória. A geração de populações é realizada iterativamente, até que um certo critério de parada seja alcançado, por exemplo, tempo de execução ou número de iterações [33].

Algoritmos genéticos são métodos heurísticos e possuem uma componente aleatória. Esta característica faz com que a busca com algoritmos genéticos seja incompleta e não-determinística. Isso significa que não é possível prever quais serão as escolhas realizadas por tal algoritmo. Além disso, para qualquer solução alcançada, se encontrada ao menos uma, não é possivel fazer qualquer tipo de afirmação sobre quão boa é tal solução. Entretanto, a busca é feita orientada a um ou mais objetivos de forma direcionada, assim, existe geralmente uma convergência da busca para uma solução boa. Existe, porém, a possibilidade de uma geração inicial levar a busca de um ótimo local, que pode ser considerada uma solução globalmente ruim. Tal problema pode ser resolvido por métodos de *niching*, que por definição, é a divisão da população em conjuntos disjuntos, de forma que a busca seja realizada cobrindo uma região mais larga do conjunto de soluções possiveis [39].

Por outro lado, o objetivo dos algoritmos combinacionais de busca é a execução determinística da mesma, sendo simples de provar o progresso de uma busca. Existem duas maneiras de realizar essa busca, por largura ou por profundidade. Na busca por largura toda a árvore de possibilidades pode ser criada inicialmente, para depois ser feita a busca de maneira exaustiva, ou para cada nível da árvore de busca é feita uma comparação incompleta, podendo resultar inclusive na poda de alguns nós. Na busca por profundidade, a busca é realizada até um nó folha, de onde é continuada a busca de maneira recursiva. É facil notar que a performance dessa busca é limitada pelo tamanho do domínio de busca, dada a natureza exaustiva destes algoritmos. Porém, se uma busca é executada completamente, o conjunto de soluções é o melhor conjunto possível para a instância do problema que foi resolvido. Existe também o caráter determinístico já que, normalmente, não existe nenhuma etapa aleatória em tais algoritmos [23].

Mesmo com as estratégias de escalonamento para o ambiente distribuído supradescritas, ainda não era possível definir qual abordagem seria utilizada no algoritmo proposto neste trabalho. Para tal foi necessário um estudo da bibliografia de algoritmos de escalonamento, que indicou quais as reais vantagens de cada abordagem e o impacto de diversas decisões tomadas em cada trabalho analisado. Assim, a proxima seção apresenta alguns trabalhos relacionados ao algoritmo proposto neste trabalho.

2.4.1 Trabalhos Relacionados

Uma extensiva revisão bibliografica foi realizada com o objetivo de buscar abordagens já realizadas para o problema de escalonamento no ambiente de nuvens. O primeiro trabalho analisado foi o de Sotiriadis et al. [41], que compilaram 18 trabalhos de metaheurísticas (i.e. heurísticas independentes do problema) para sistemas distribuídos. Desse trabalho, foi possível estudar as características principais que os trabalhos referenciados por Soti-

riadis et al. apresentaram. Das características mais comuns para o ambiente distribuído tem-se flexibilidade, escalabilidade e interoperabilidade. Com exceção da escalabilidade, o algoritmo proposto neste trabalho não tem como responsabilidade prover tais caracterísicas. Assim, foram identificadas também duas características que a maioria dos trabalhos analisados por Sotiriadis et al. são desprovidos, o qual é o uso de dados de execução colhidos em tempo real, e um sistema de histórico. Dessa forma, decidiu-se então que essas características seriam utilizadas no algoritmo proposto neste trabalho, dada a natureza do ambiente de nuvem.

Além disso, três trabalhos usando algoritmos genéticos foram analisados. O primeiro, de Yu e Buyya [46], tem como objetivo otimizar custo e tempo de execução. Para isso, foi usada uma equação que combina de maneira aritmética ambos os objetivos. Também foram definidos parâmetros para os mecanismos de *crossover* e mutação. Todavia, embora esse algoritmo tenha tido uma rápida convergência, não foi mencionado o tempo de execução do mesmo, e como mencionado anteriormente, não existe uma forma de provar quão boa é uma solução em nível global.

O próximo trabalho analisado, de J. Liu et al. [28], propõe um algoritmo genético semelhante ao de Yu e Buyya [46], tendo como diferenças mais marcantes as funções objetivo e a maneira como esses objetivos são compilados. Agora, os objetivos são o consumo energético do sistema e o lucro, obtido pela diferença entre o custo de execução cobrado e o custo necessário, pago ao provedor de recursos. Como metodologia de combinação de objetivos foi usada uma frente de pareto bi-dimensional [44], que apresentou melhores resultados quando comparada com a abordagem de Yu e Buyya [46].

Por fim, o último trabalho de algoritmo genético investigado foi de Kessaci et al. [25]. Este por sua vez, usa pareto [44], assim como Liu et al. [28], tendo três objetivos de otimização, os quais são lucro, consumo energético e emissão de CO2. Além disso, também foi feito um comparativo com outros nove trabalhos de escalonamento, no qual foi visto que somente a abordagem de Kessaci et al. [25] usa pareto.

Os trabalhos [29, 30, 40] usaram busca combinacional como mecanismo de busca. Mais adiante, a busca combinacional não foi implementada em nenhum deles, sendo usadas bibliotecas para o paradigma de programação CP (*Constraint Programming*) [35]. Esse paradigma baseia-se na descrição de um problema como um conjunto de variáveis e restrições para os valores possíveis para as mesmas. A partir dessa descrição é realizada uma busca combinacional após a propagação dessas restições, de maneira iterativa até o momento em que as soluções são encontradas. Vale ressaltar que a etapa de busca ocorre, então, apenas para os valores das variáveis que possam produzir resultados válidos. Essa abordagem é de complexidade exponencial, sendo baseada no número de variáveis e restrições usadas. Por exemplo, uma das maneiras mais simples de implementação de

um escalonador de t tarefas a r recursos, usando CP, é a criação de uma matriz $t \times r$, acrescido das restrições impostas a conjuntos de variáveis. Dessa forma, o crescimento no valor de qualquer uma das variáveis piora exponencialmente o tempo de busca devido à relação variavel - restrição [35].

Assim, têm-se como vantagens no uso de CP a facilidade de representação do problema e o modelo de restrições. Assim, CP deve ser usado apenas quando o modelo de restrições é suficientemente complexo para o seu uso ser justificado. Os trabalhos [29, 30, 40] usaram CP como uma maneira de simplificar a parte de dependência de tarefas, podendo assim concentrar esforços na otimização de objetivos. Em um primeiro momento foi pensado no uso de CP neste trabalho. Porém, dado que o escalonador a ser apresentado no Capítulo 4 escalona apenas tarefas independentes, não existem restrições suficientes para legitimar o uso de CP.

Com o conhecimento adquirido pelo estudo das abordagens descritas anteriormente, foi possivel redigir cuidadosamente a ponderação de vantagens e de desvantagens de cada abordagem. Dessa forma, o algoritmo proposto neste trabalho, a ser apresentado no próximo capítulo, considerou ideias de varios dos trabalhos analisados.

2.5 Considerações Finais

Este capítulo introduziu os conceitos básicos necessários para a definição do ambiente de nuvens federadas, o qual este trabalho tem como base. Além disso, os conceitos fundamentais para a definição do algoritmo de escalonamento proposto neste trabalho foram descritos e serão necessários para a compreensão de decisões tomadas na concepção do mesmo, visto em detalhes no próximo capítulo.

Capítulo 3

Escalonador de Tarefas Proposto

Este capítulo descreve o algoritmo de escalonamento proposto, demonstrando também partes de sua implementação. Inicialmente, o capítulo apresentará algumas considerações iniciais, explicitando os objetivos que moldaram o algoritmo. Em seguida, é explicado, de maneira superficial, como os objetivos mencionados são alcançados, apresentando também alguns termos básicos para o entendimento do algoritmo. Finalmente, a implementação do algoritmo proposto é detalhadamente exposta.

3.1 Visão Geral

O algoritmo de escalonamento proposto neste trabalho tem como objetivos ser *any-time* e incremental com uma rápida convergência para boas soluções. Por incremental e *any-time*, entende-se que o algoritmo convergira para um conjunto de soluções melhor com o decorrer de sua execução, e que o algoritmo poderá ser parado em qualquer momento da sua execução, retornando ainda um conjunto de soluções.

Os objetivos de otimização são a minimização de ambos o tempo e o custo total (monetário) de execução de um *pipeline*. Todavia, levando-se em consideração a existência de dois objetivos de otimização, é improvável que haja apenas uma solução ótima, dada a natureza multimodal do problema.

Para cumprir as características destacadas acima, o algoritmo proposto levou em consideração a natureza das tarefas a serem escalonadas. Assim, a natureza das tarefas a serem executadas pelo BioNimbuZ é extremamente heterogênea, com tempos de execução podendo variar de poucos minutos a horas ou dias, e possuir uma demanda de memória com ordem variando de megabytes a dezenas de gigabytes [38]. Além disso, tarefas de bioinformática possuem como requisito, em alguns casos, a necessidade de acesso à bancos de dados.

Além do exposto acima, assim como qualquer outra política de escalonamento para o BioNimbuZ, esta deve obedecer a interface do serviço de escalonamento da camada de núcleo do BioNimbuZ, ou seja, receber como parâmetros um conjunto de tarefas independentes que devem ser executadas prontamente, e um conjunto de *peers*, responsáveis pela execução dessas tarefas. Estes *peers* serão tratados doravante como recursos, e cada recurso é definido de acordo com suas características de hardware na qual a máquina virtual está executando.

Das características dos recursos usadas pelo algoritmo proposto se destacam a frequência de *clock* do processador e o custo da instância por hora. É valido ressaltar que o número de recursos disponíveis para o escalonamento é da ordem de $20 - 100$ recursos. Isso ocorre devido à diversidade de nuvens que podem ser utilizadas (por exemplo: Amazon [2], Azure [6], Google [10], etc), e pela própria diversidade de instâncias disponibilizadas por cada um desses provedores.

3.2 Estratégia de Escalonamento

Por definição, este algoritmo deve retornar para cada instância do problema uma função sobrejetiva representando um estado de alocação. Essa função tem como domínio o conjunto de tarefas independentes a serem alocadas e como imagem o conjunto de recursos disponiveis. A partir disso, foi projetado então um algoritmo usando busca combinacional guiada por um algoritmo de ordenação de busca. Por ser uma busca combinacional é garantida completude no sentido de optimalidade por força bruta (i.e. dado que todas as soluções são buscadas todas as melhores soluções são encontradas), e pela busca ser guiada via ordenação de valores de escolha (i.e. ordenação de nós de busca), existe uma maior chance de encontrar boas soluções globais com um menor tempo de execução, sendo essa performance dependente do algoritmo de ordenação [37].

Outra característica dos algoritmos de busca combinacional completa é a explosão combinacional. Dado que o grande número de recursos disponíveis para escalonamento é um dos responsáveis pela explosão combinacional, foi usado o *beam search* [38] como uma heurística para diminuir a complexidade da execução. O *beam search* é por definição uma heurística que visa reduzir o número de possibilidades a serem buscadas combinacionalmente a partir da poda da árvore de busca. Esta poda é dada por uma valor definido como *beam*, que indica o número máximo de nós filhos a serem visitados, dado um nó pai. A escolha, normalmente, é feita a partir de um algoritmo de ordenação, onde os n primeiros filhos são escolhidos, sendo n o tamanho do *beam*. Todavia, o problema dessa abordagem é a perda de completude.

Além disso, neste trabalho foi usado como algoritmo de ordenação para o *beam search*, e método de compilação de objetivos, uma frente de pareto. Frente de pareto é definida por Veldhuizen e Lamont [44] como, dado um conjunto de soluções S em um espaço n-dimensional, onde n é o número de objetivos a serem otimizados, a frente de pareto é o subconjunto S' de S, onde nenhum ponto de S' é estritamente dominado por qualquer outro ponto de S. Um ponto *s1* é estritamente dominado por *s2* caso, para os n objetivos a serem otimizados, *s2* possua melhores valores para todos os n objetivos quando comparado com *s1* [31] [44].

Dessa maneira, dado que este trabalho tenta otimizar apenas dois objetivos (funções 3.1 e 3.2 a serem minimizadas, que serão explicadas em detalhes adiante), foi implementado um algoritmo para montagem de uma frente de pareto bidimensional, conforme observado no Algoritmo 1. Nessa implementação, todos os pontos são ordenados de acordo com um dos objetivos, sendo escolhido neste caso o tempo máximo de execução, visto na linha 4 do algoritmo 1. Vale ressaltar que qualquer um dos objetivos poderiam ser utilizados, retornando a mesma frente de pareto, e que a ordenação foi do melhor tempo ao pior (i.e. menor tempo ao maior). Em seguida, são selecionados os pontos que possuem dominância do segundo objetivo (custo) sobre o ponto anterior. Essa dominância é definida como, dado dois pontos A e B, B sendo o sucessor de A, B domina A, se e somente se, para o objetivo avaliado, o valor de B for melhor do que o de A. No caso do custo, essa dominância seria B possuir um custo menor do que A. Esta etapa é ilustrada pela comparação realizada na linha 7 do Algoritmo 1.

Dessa forma, apenas os nós que são completamente dominados por, pelo menos, um dos pontos na frente de pareto são removidos. Outra característica importante é que todos os pontos em uma frente de pareto são ditos igualmente bons, já que não existe um ponto que domine estritamente outro da mesma frente.

Algorithm 1 Algoritmo de Montagem de uma Frente de Pareto Bidimensional

```
1:  function PARETO_CURVE(resources)
2:      limit ← ∞
3:      paretoCurve ← ∅
4:      sortedRes ← sortByMaxTime(resources)
5:      for all resource ∈ sortedRes do
6:          execCost ← resource.getExecCost()
7:          if execCost < limit then
8:              paretoCurve.add(r)
9:              limit ← execCost
10:         end if
11:     end for
12:     return paretoCurve
13: end function
```

3.3 Descrição Detalhada do Algoritmo Proposto

Para otimizar a busca, de maneira a agilizar a descoberta de boas soluções, foi escrito um algoritmo dividido em três estágios, nos quais haveriam dois estágios de busca preliminares ao *beam search*. O primeiro estágio foi concebido como uma busca de uma única solução de maneira gulosa, ou seja, realizando as melhores escolhas locais na expectativa de que esta solução, que é retornada rapidamente, seja boa globalmente. O segundo estágio foi pensado baseando-se no trabalho proposto por Harvey e Ginsberg [24], tendo como objetivo expandir o conjunto de soluções de maneira rápida, porém, mais lentamente quando comparado com o primeiro estágio. Finalmente, o terceiro estágio consiste na execução do *beam search* de maneira iterativa, resultando em um algoritmo completo [48]. Para o melhor entendimento do algoritmo proposto, cada um desses estágios é apresentado em detalhes nas proximas seções.

3.3.1 Estágio 1 - Busca Gulosa

No primeiro estágio foi realizada a busca gulosa, na qual, para cada nó é escolhido o melhor nó filho local de acordo com o algoritmo de ordenação, que será apresentado na proxima seção, e a busca é continuada a partir dele da mesma forma. Essa busca é concluída quando se chega ao nó folha. O custo de execução deste estágio é baixo, tendo complexidade linear, definida pelo número de tarefas a serem escalonadas (i.e. a profundidade da busca).

Algoritmo de Ordenação

As soluções em um conjunto de soluções em uma frente de pareto são consideradas igualmente boas como já foi dito anteriormente. Dessa forma, foi pensado em uma maneira de ordenar tais soluções baseado em um grau de preferência por algum dos objetivos. Esse gradiente indicador de preferência foi definido como um número no intervalo contínuo entre zero e um, onde um gradiente zero indica uma preferência absoluta pela otimização do objetivo tempo (menor tempo), e o gradiente igual a um indica uma preferência absoluta por custo (menor custo monetário).

Uma primeira abordagem para a definição de ordem foi usar esse gradiente de preferência de maneira estática como um vetor no plano de soluções, visto na Figura 3.1. A ordenação seria feita pela proximidade dos pontos de solução contidos no conjunto ótimo de pareto ao vetor de preferência. Porém, usando essa estratégia, o vetor nem sempre estará de acordo, logicamente, com o que era esperado do gradiente. Exemplificando, ao se escolher um valor 0.5 é esperado que a preferência seja igual entre os dois objetivos. Assim, ao comparar as Figuras 3.2 e 3.3 fica evidente que, na prática, os vetores

de preferência poderiam ser definidos como um ou zero, respectivamente, sem alterar o ordenamento final das soluções. O ordenamento das soluções nas Figuras 3.1 , 3.2 e 3.3 são do ponto P1 ao ponto P5, sendo P1 a melhor solução.

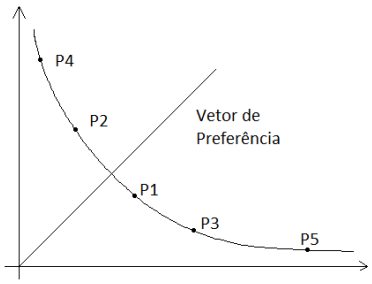


Figura 3.1: Caso Base de uma Curva de Pareto.

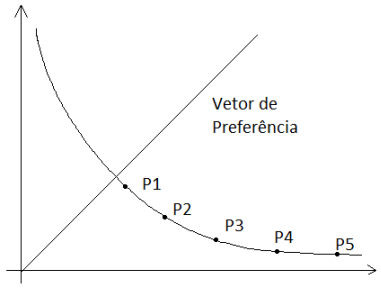


Figura 3.2: Caso com Alto Custo.

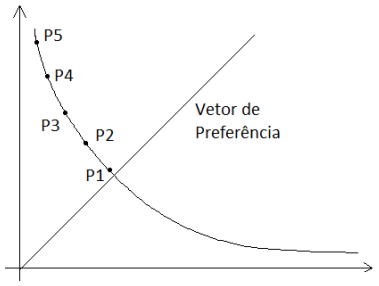


Figura 3.3: Caso com Alto Tempo de Execução.

Dessa forma, o ordenamento de soluções do conjunto ótimo de pareto foi implementado de acordo com o posicionamento em um *array* ordenado, descrito pelo Algoritmo 2, tendo como entrada uma curva de pareto e um gradiente de preferência com valor entre 0 e 1, retornando uma lista com todas as soluções ordenadas. Essa alteração faz com que o ordenamento dos pontos sejam igual nos três casos apresentados acima, independente do posicionamento do conjunto de pontos, quando comparado com o vetor de preferência.

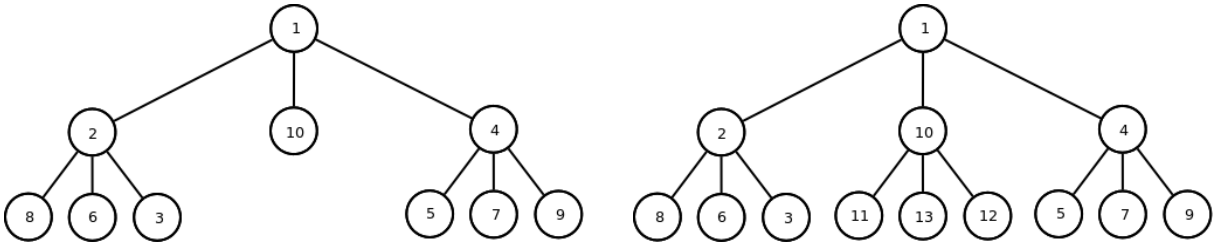
Como foi definido que para o gradiente com valor um, a preferência seria absoluta para o custo, o *array* será ordenado pelo custo de maneira crescente, visto na linha 2 do Algoritmo 2. A seguir, as soluções serão ordenadas pela proximidade de seus índices ao produto do gradiente de preferência pelo tamanho do conjunto de pareto, descrito na linha 6 do algoritmo.

Algorithm 2 Algoritmo de Ordenação de Soluções em uma Curva de Pareto

```
1:  function SORT_PARETO(pareto_frontier, preference)
2:      pre_sorted ← sortByCost(pareto_frontier)
3:      sorted_pareto ← ∅
4:      while pre_sorted ≠ ∅ do
5:          length ← length(pre_sorted)
6:          pos ← length × preference
7:          sorted_pareto.add(pre_sorted.remove(pos))
8:      end while
9:      return sorted_pareto
10: end function
```

Estruturas de Busca

Para implementar este estágio, primeiramente, foi definida a estrutura de um nó de busca. Este nó é composto pelo estado atual da busca e pelo conjunto de nós que serão buscados subsequentemente, sendo dividido em três subconjuntos: nós a serem visitados, nós sendo visitados e nós que já foram completamente visitados. Cada nó também possui uma lista de recursos a serem usados no escalonamento, sendo que cada recurso pode ter nenhuma, uma ou mais tarefas alocadas a ele.



(a) Árvore Antes do Nó 10 Ser Visitado. (b) Árvore Depois do Nó 10 Ser Visitado.

Figura 3.4: Exemplos de Árvores de Busca.

As Figuras 3.4a, 3.4b e 3.5 exemplificam a estrutura da árvore de busca para três recursos e duas tarefas a serem escalonadas, na qual cada nó é representado por um valor numérico correspondente à ordem de busca. A criação de um nó representa a alocação de uma tarefa a um recurso, e que a partir do novo nó deverão ser buscadas todas as possibilidades subsequentes de alocação da próxima tarefa, até que não haja mais tarefas. No exemplo da Figura 3.4a o nó 1 é o nó raiz, no qual nenhuma tarefa foi escalonada. Os seus filhos 2, 10 e 4 correspondem à alocação da tarefa *T1* nos recursos *R1*, *R2* e *R3*, respectivamente. Após a criação de nós, é escolhido um dos nós criados para prosseguir a busca. Este nó, caso seja um nó que nunca foi visitado antes (classificado assim como um nó a visitar), cria todos os seus nós filhos nessa primeira vez que é visitado. Este caso é ilustrado nas Figuras 3.4a e 3.4b, onde o nó 10 ainda não havia sido visitado na Figura 3.4a, e após ser visitado pela primeira vez, são criados os nós 11, 12 e 13, vistos na Figura 3.4b.

Como mencionado anteriormente, para cada nó filho é escolhido um recurso diferente entre esses nós filhos, no qual a próxima tarefa a ser escalonada é atribuída. Dessa maneira, todas as possibilidades de alocação são levadas em consideração. Os novos nós filhos são então definidos como nós a serem visitados, e assim que são visitados pela primeira vez, são movidos para a lista de nós sendo visitados, no nó pai.

Assume-se como exemplo que na Figura 3.5 a primeira solução seja oriunda da busca realizada pelos nós 1, 2 e 3. Ao começar a busca no nó 1, os nós 2, 10 e 4 são criados, onde a tarefa *T1* é alocada nos recursos *R1*, *R2* e *R3*, respectivamente. Todos os três nós são

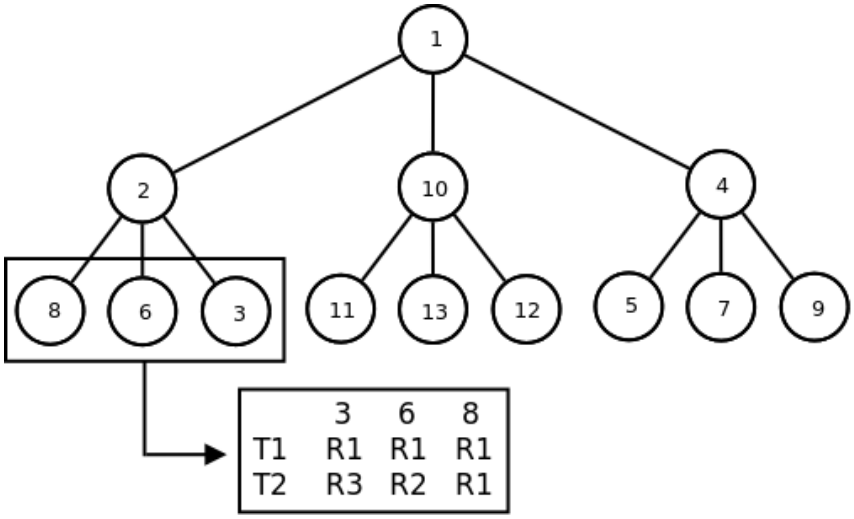


Figura 3.5: Foco no Estado de Escalonamento de Três Nós Folha.

definidos como nós a visitar. Em seguida, assumindo a escolha do nó 2, este é dito como nó sendo visitado para o nó 1, e, subsequentemente, os seu nós filhos 3, 6 e 8 são criados e definidos como nós a visitar. Finalmente, ao visitar o nó 3, não existem mais tarefas a serem escalonadas, tornando o nó 3 um nó folha, e dessa forma, sendo considerado um nó visitado pelo nó 2. Como consequência, assim que todos os nós filhos de um nó são exauridos (i.e. estão na lista de nós visitados), ou não possuem nenhum nó filho, por exemplo, o nó 3 da Figura 3.5, este é definido como nó visitado pelo nó pai. Esses nós folhas ao serem visitados geram uma solução final, ou seja, uma solução na qual todas as tarefas foram escalonadas. A Figura 3.5 mostra o estado de escalonamento dos três nós 3, 6 e 8. Esses estados são ditos completos pois todas as tarefas já foram escalonadas, gerando assim soluções. Essas soluções são, então, armazenadas em uma lista de soluções finais, definidas também como uma frente de pareto, podendo ser removidas da lista de soluções finais caso sejam completamente dominadas por outras soluções, garantindo assim que apenas as melhores soluções buscadas serão mantidas.

Independente do estágio do algoritmo, a escolha dos nós a serem visitados é ordenada de acordo com o algoritmo de ordenação que será explicado adiante na proxima seção. Outro ponto que é válido ressaltar é que, como a árvore de busca é persistente em memória, a busca nunca realiza as operações de cálculo mais de uma vez. Além disso, nenhum nó folha será buscado mais de uma vez e nenhum nó intermediário, que já tenha concluído a busca em seus nós filhos, será visitado novamente.

Outra estrutura presente em cada nó de busca é a lista de recursos. Desta lista são extraídos os valores dos objetivos tempo e custo de execução, até o ponto de busca desse nó. Cada recurso possui, além da sua capacidade de processamento e custo por hora, uma lista de tarefas alocadas a ele. O tempo de execução de uma lista de recursos é definido

como o maior tempo de execução de todos os recursos, e o de custo é a soma de todos os custos individuais de cada recurso. Os cálculos de tempo e custo de execução para cada recurso estão definidos pelas Equações 3.1 e 3.2, sendo estes o tempo máximo de execução de todas as tarefas alocadas a este recursos, e o custo total para a execução dos mesmos. Nestas equações, *task.cycles* representa uma alegoria ao número de ciclos necessários para a execução de uma tarefa, *res.frequency* é a frequência do processador de um dado recurso, e *resource.cost* é o custo para o uso de um recurso em um período de tempo. Vale ressaltar que *task.cycles* possui um valor normalizado entre tarefas, representando o custo computacional de uma tarefa, semelhante ao número de ciclos de clock.

$$maxTime_{res} = \forall task \in res \rightarrow max\left(\frac{task.cycles}{res.frequency}\right) \tag{3.1}$$

$$totalCost_{res} = \sum_{task \in res} \frac{res.cost \times task.cycles}{res.frequency} \tag{3.2}$$

3.3.2 Estágio 2 - *Limited Discrepancy Search*

De acordo com Harvey e Ginsberg [24], um possível motivo para uma heurística falhar é um pequeno número de escolhas erradas, definidas por ele como "*wrong turns*". Isso ocorre caso seja criado um novo *branch* de busca, na qual tenha havido uma discrepância entre a escolha da heurística e a melhor escolha. Como não é possível saber *a priori* quais escolhas serão discrepantes com as melhores, é assumido que cada escolha que foi realizada pode ter sido errada. Partindo desse trabalho, o mesmo foi adaptado para o domínio de busca combinacional. Dessa forma, para as n tarefas a serem escalonadas e para a árvore de busca oriunda do Estágio 1, que possui apenas um caminho de busca até um nó folha, serão realizadas n outras buscas até um nó folha, sendo que, cada busca chegará a um nó folha diferente, gerando assim uma solução final nova. Este estágio verifica n novas soluções, já que para qualquer árvore de busca com n tarefas a serem alocadas, devem ser feitas n escolhas de recursos para a busca chegar a um nó folha.

O algoritmo de busca desse estágio é visualizado na Figura 3.7. Como mencionado anteriormente, a busca começa do resultado do primeiro estágio, na Figura 3.6. A partir dela, nos nós 1 e 2 serão realizadas as buscas com as segundas melhores escolhas locais, 4 e 6, respectivamente. Isso é visto na Figura 3.7, na qual os caminhos hachurados são os novos caminhos buscados, retornando mais duas soluções, dos nós 5 e 6.

3.3.3 Estágio 3 - *Beam Search* Iterativo

O algoritmo de *beam search* clássico, definido por Reddy em [38], é um algoritmo de busca combinacional onde é usado um algoritmo de ordenação de soluções parciais em conjunto

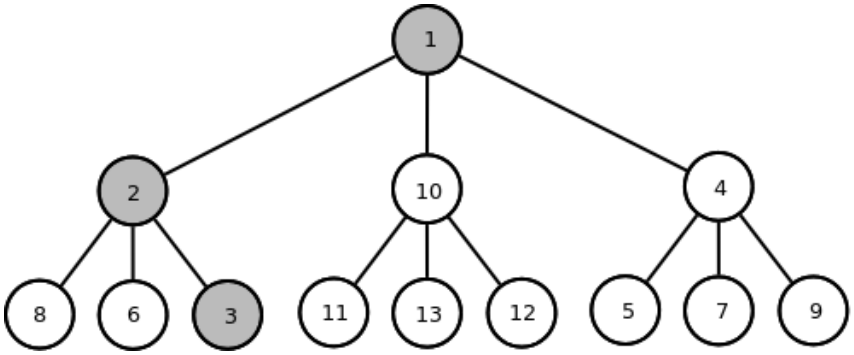


Figura 3.6: Exemplo de Árvore de Busca para Abordagem Gulosa.

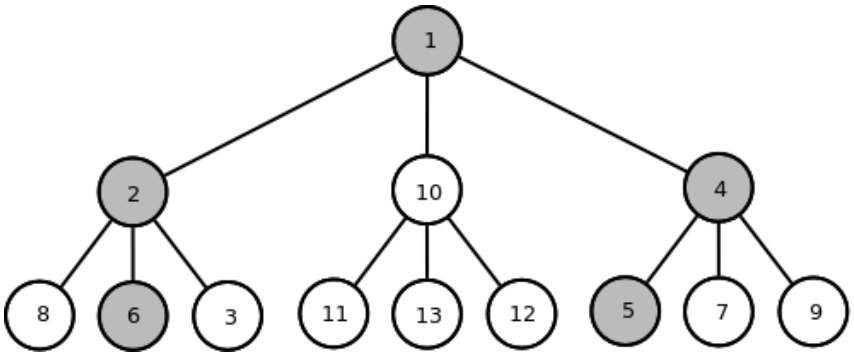


Figura 3.7: Exemplo de Árvore de Busca para o Algoritmo *Limited Discrepancy Search*.

a uma heurística de remoção de nós de busca, baseado na ordem. A quantidade de nós removidos (ou nós a serem buscados) é definida pelo usuário do algoritmo. Este algoritmo reduz consideravelmente a complexidade de uma busca. Porém, com essa redução é perdida a característica de completude. Esta é definida, tal que, dado uma busca que chegou ao fim de sua execução, o conjunto de soluções é dito ótimo, pois todas as possibilidades foram buscadas, ou as que não foram buscadas são provadas que nunca entrariam para o conjunto de soluções ótimas.

O trabalho de Zhang [47] propõe a execução iterativa do *beam search*, enfraquecendo a heurística de poda a cada iteração. Esta heurística, porém, é definida como um custo parcial máximo que cada nó pode ter, quando calculada a diferença entre ele e seu nó pai. Esta estratégia gera então uma árvore de busca com largura variável, ao contrário do algoritmo proposto por Reddy [38].

O algoritmo usado no terceiro estágio, inspirado pela compinação dos algoritmos [47] e [38], descrito no Algoritmo 3, e proposto neste trabalho, apresenta uma maneira original de executar o algoritmo *beam search* de maneira completa e multiobjetivo. O *beam search* é executado de maneira iterativa, assim como no trabalho proposto por Zhang [47]. Entretanto, a heurística de seleção de nós permanece fiel à estratégia clássica [38], começando com o *beam* de tamanho dois, e indo até o tamanho máximo (i.e., o número

total de recursos disponíveis).

Algorithm 3 Algoritmo de *Beam Search* com Poda

```
1: function BEAM_SEARCH(node, beam, tasks, tasksRemaining)
2:     needed ← beam-(node.visiting.length+node.visitedCount)
3:     if needed > 0 then
4:         for i = 0 → needed do
5:             nextNode ← node.toVisit.poll()
6:             node.visiting.add(nextNode)
7:             if tasksRemaining > 1 then
8:                 nextNode.toVisit ← generatePriorityQueue(jobs.get(tasks.length()−
   tasksRemaining), nextNode.rl)
9:             end if
10:        end for
11:    end if
12:    for all n ∈ node.visiting do
13:        if n.visiting.length()+n.toVisit.length() > 0 then
14:            if n.prunable AND (n.time() > node.maxTime() OR n.cost() >
   node.maxCost()) then
15:                node.visiting.remove(n)
16:            else
17:                node.addToPareto(BEAM_SEARCH(n, tasks, tasksRemaining − 1))
18:            end if
19:        else
20:            if tasksRemaining = 1 then
21:                updateBest(n)
22:                node.addToPareto(n.rl.maxTime(), n.rl.fullCost())
23:            end if
24:            node.visiting.remove(n)
25:        end if
26:    end for
27:    return node
28: end function
```

O algoritmo é dividido em duas etapas. No primeiro momento, a lista de nós sendo visitados é expandida para condizer com o tamanho do *beam*. Esta etapa é vista nas linhas 2 a 11 do Algoritmo 3. Como será visto na próxima etapa, apenas nós na lista de nós sendo visitados são percorridos. Para um nó estar de acordo com o tamanho do *beam*, a soma de nós visitados com nós sendo visitados, deve ser igual ao tamanho do *beam*, como visto nas linhas 2 e 3 do Algoritmo 3. Esta correção da lista de nós sendo visitados é realizada pela transferência de nós da lista de nós a serem visitados, para a lista de nós sendo visitados. Estes nós não possuem nós filhos, já que esta será a primeira vez que estes serão visitados. Os nós filhos serão então gerados nesta etapa, representado na linha 8. Na geração desses nós, cada um deles recebem a tarefa a ser alocada em recursos

diferentes entre eles, em seguida, sendo compilados em uma curva de pareto. Os nós que não entrarem na curva recebem em sua variavel *prunable* o valor *true*. Finalmente, todas as soluções são ordenadas de acordo com o Algoritmo 2, sendo os nós ditos como *prunable* sendo ordenado da mesma forma após a lista de nós pertencentes à curva de pareto.

A segunda etapa é a busca em si, realizada recursivamente. Nela, para cada nó, são realizadas n buscas recursivas em n nós pertencentes à lista de nós sendo visitados, sendo n a largura do *beam*, visto na linha 12 do Algoritmo 3. Caso o nó sendo visitado seja um nó folha, este é dito uma solução completa e é colocado na lista de soluções finais (linha 22), e depois é movido da lista de nós sendo visitados para a lista de nós visitados. A busca é realizada até que todos os nós folhas sejam buscados, ou que o algoritmo tenha esgotado seu tempo máximo de execução.

Sabe-se que o algoritmo *beam search* remove permanentemente da árvore de busca nós, ditos desnecessários para encontrar uma boa solução. Isto torna o algoritmo incompleto, no sentido de que para uma certa solução dita ótima, é possível que tal solução nunca seja encontrada. Como foi proposto anteriormente, a execução do *beam search* usando enfraquecimento iterativo na variável *beam* deve tornar o algoritmo *beam search* completo [47]. Se esse enfraquecimento do *beam* chegar ou seu limite máximo (i.e. *beam* igual à largura máxima da árvore), nenhum nó será removido da busca por essa heuristica. Dessa forma, ao *beam* do algoritmo de busca iterativa tender ao valor máximo da largura da arvore, este algoritmo pode ser dito completo nó sentido de optimalidade.

Poda de Árvore de Busca com Pareto Generalizada

Contudo, embora os dois primeiros estágios sejam executados quase que instantaneamente para os casos alvo, o terceiro estágio leva um tempo considerável para ser executado completamente, normalmente, consumindo a maior parte do tempo. Ademais, nem sempre o número de soluções retornadas, ou a localização da última solução em relação ao *beam* que foi encontrada, validam a quantidade de tempo gasto pelo mesmo. Consequentemente, torna-se necessária uma otimização da busca, dado que é esperado um alto número de recursos, resultando em uma árvore de busca com grande largura.

Para superar essa limitação foi pensada uma estratégia de poda generalizada para uma árvore de busca combinacional que usasse pareto como objetivo. A poda proposta neste trabalho foi realizada em nós de busca, nos quais fosse impossível se chegar a uma solução aceitável, isto é, que entrasse no conjunto de pareto final de soluções.

Na Figura 3.8 tem-se um exemplo da execução de um nó de busca com apenas três nós filhos a, b e c. Esses por sua vez possuem os nós filhos d, e e f respectivamente, sendo estes, nós folha. É tentado então podar o nó c caso não exista nenhuma boa solução

oriunda dele. Tem-se no eixo das abscissas o tempo máximo de execução para cada nó de busca, e no eixo das ordenadas o custo total da cada nó.

Teorema: Não existe nenhum nó f oriundo de c, que seja pertencente ao conjunto ótimo de pareto de d, e e f se c.c >= d.c ou se c.t >= e.t.

Prova: Usando como modelo a Figura 3.8, para um nó c não ser podavel deve existir um nó f pertencente à curva de pareto dos nós d, e e f. Os nós d e e são os nós com o maior custo oriundo de a e com o maior tempo de execução oriundo de b respectivamente. Pelo Algoritmo 1, a partir de uma curva de pareto com os nós d e e, um nó f somente pertenceria a esta curva se e somente se f.c < d.c ou .t < e.t, sendo x.c o valor do objetivo custo de um nó x e y.t o valor do objetivo tempo de execução de um nó y. Dessa forma, se c.c >= d.c ou se c.t >= e.t, f nunca será uma solução valida, quando existirem d e e, sendo assim c um nó podavel. ■

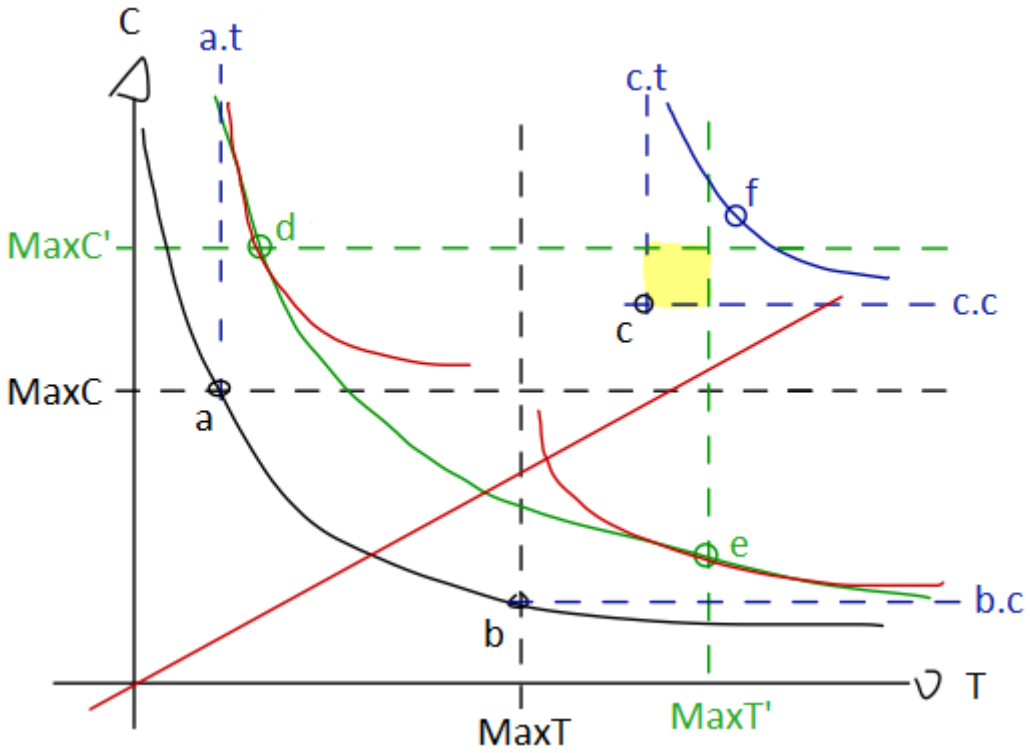


Figura 3.8: Caso de Duas Tarefas Alocadas a Três Recursos no Qual o Nó c Pode Ser Podado.

A implementação dessa estratégia pode ser vista no Algoritmo 3. Esta foi feita nas linhas 14 a 17, e é definida por duas condições. Primeiramente, um nó deve ser podável. Isso significa ser um nó não pertencente à curva de pareto do seu nó pai. Em seguida, são verificadas as condições de tempo e/ou custo, descritas previamente.

Em relação à completude do algoritmo de busca usando essa poda, ela se mantém. Isto se deve ao fato de que, como foi provado anteriormente, nenhum nó que seja pertencente

ao conjunto ótimo de pareto, ou que seja antecessor a um desses nós, é removido. Dessa forma, pode-se concluir que o algoritmo permanece completo.

Uma limitação desse algoritmo de poda, porém, é a necessidade de pelo menos dois nós não podáveis/podados para existir a possibilidade de poda de um terceiro nó. Ademais, dada a natureza da poda, que ocorre apenas quando um nó realiza uma escolha ruim que inviabiliza esse nó, é esperado que tal caso seja mais comum nos níveis mais inferiores da árvore de busca, e mesmo assim, que não seja um evento tão comum. Essas expectativas reduzem os casos onde a poda seria efetiva.

Além disso, foi visto empiricamente que tais casos foram suficientemente frequentes para impactar fortemente a busca, tornando-a viável, e até eficiente, em certos casos. Também foi identificado que, devido à limitação de pelo menos três nós para ser possível a poda, este algoritmo de poda é inútil caso a busca não progrida ao ponto no qual o *beam* é maior ou igual a três. O algoritmo de poda torna-se ineficiente quando a árvore de busca tenha profundidade alta, sendo, dessa forma, a chance de poda de um nó diretamente proporcional à profundidade do mesmo.

3.4 Considerações Finais

Neste capítulo foi descrito em detalhes os objetivos do algoritmo proposto e como tais objetivos foram alcançados. Entre eles, foi implementado um algoritmo baseado no algoritmo *beam search*, em três estágios, com um algoritmo de ordenação usando uma frente de pareto, e usando uma estratégia de poda, para diminuir o tempo de busca enquanto é mantida a completude. Foram vistas, também, limitações para o algoritmo proposto. Dada a natureza combinatória do algoritmo proposto, o tempo necessário para a execução completa do mesmo para *batches* com um grande número de tarefas torna o algoritmo ineficiente para esses casos. No próximo capítulo será mostrado esse limite, entre outros resultados, e no Capítulo 5 serão discutidas maneiras de lidar com esse problema.

Capítulo 4

Testes e Resultados

No primeiro momento este capítulo abordará as métricas e os comportamentos a serem observados a partir do algoritmo de escalonamento proposto no capítulo anterior. Em seguida, serão apresentados os testes realizados e o ambiente de execução para o mesmo. Finalmente, todos os testes serão discutidos em detalhes, acompanhados de gráficos do tipo *box plot*, dos quais serão extraídas algumas análises importantes.

4.1 Objetivos dos Testes

Ao testar qualquer algoritmo, duas métricas se destacam, a qualidade dos resultados e o tempo necessário para eles serem alcançados. Dada a natureza do algoritmo, que foi combinacional, e a garantia de completude, no sentido de optimalidade que foi discutido no capítulo anterior, os resultados são garantidos como sendo os melhores havendo tempo suficiente para a execução completa, no sentido de progresso da busca. Para tal, também é necessário que a estratégia de poda não remova nenhum resultado possivelmente bom, o que também foi provado no capítulo anterior.

Outro comportamento importante de ser testado é a sensibilidade do algoritmo às variações nos tamanhos das entradas, ou seja, número de tarefas a serem escalonadas e número de recursos a serem utilizados. Tal teste é necessário devida a natureza da complexidade desse problema de escalonamento, que é NP-Completo [36].

4.2 Ambiente de Testes

Para a análise do algoritmo proposto foram realizados testes em um ambiente simulando o ambiente de nuvens federadas, usando um conjunto extensivo de tarefas de diversos custos, demonstrando assim a reatividade do algoritmo para entradas de diferentes tamanhos.

Todos os testes apresentados neste capítulo foram realizados em dois ambientes, o primeiro para os testes de performance era composto por 2 maquinas contendo cada uma 1 processador Intel i7-3770 e 8 GB de RAM, e o segundo, para os testes de reatividade contendo sendo um ambiente virtualizado, usando o *hypervisor* Xen [14], em 1 processador Intel i7-3779. O ambiente virtual disponibilizado contava com 2 núcleos e 2 GB de RAM. O sistema operacional virtualizado foi o Ubuntu 15.10. Por fim, é importante destacar que todos os testes foram realizados em ambiente exclusivo, ou seja, não havia nenhum outro processo executando concorentemente.

4.3 Testes de Performance

Esta bateria de testes foi concebida para comparar o algoritmo proposto com a política de escalonamento anterior do BioNimbuZ, chamada ACOSched [42]. Essa política é uma adaptação do algoritmo *Ant Colony Optimization* (ACO). Este algoritmo baseia-se na premissa que, para uma colonia de formigas encontrar um caminho ótimo entre dois pontos varios caminhos serão escolhidos, e ao serem atravesados cada formiga libera feromônios neste caminho. Uma formiga, ao escolher um caminho, dá preferência a camhinhos com maior quantidade de feromônios. Essa escolha é baseada em uma equação que descreve a probabilidade de escolha entre os possiveis caminhos, sendo esta diretamente proporcional à quantidade de feromônios. Dessa forma, o algoritmo se adapta ao ambiente no qual é implementado [42]. O ACOSched usa desta abordagem, usando em sua função objetiva os feromônios de cada caminho (recurso escolhido para escalonamento) e a latência dos recursos com o nó mestre do BioNimbuZ, otimizando apenas o tempo de execução.

O teste comparativo consiste na execução paralela de três *pipelines* iguais de tarefas reais de Bioinformática. Os *pipelines* eram compostos por uma tarefa do software de alinhamento Bowtie [7], seguida de um *script* de conversão *sam* para *bed* em Perl, terminando com um outro *script* em Perl para geração de intervalos a partir de genomas, chamado *genome2interval*, conforme apresentado na Figura 4.1. É valido ressaltar que embora as tarefas ordenadas por precedencia temporal, uma tarefa a ser executada não pode possuir nenhuma dependência sendo executada ou a ser executada. Caso exista uma ou mais tarefas pendêntes a tarefas com dependências deverá ser colocada em espera, sendo executada apenas quando as mesmas finalizarem as suas execuções.

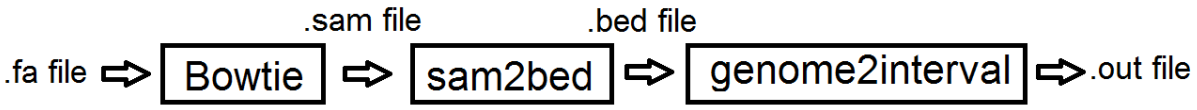


Figura 4.1: Pipeline Usado Para Teste de Performance.

Foram realizadas duas baterias de cinco testes, uma bateria para cada escalonador. Foram medidos os tempos de execução total para cada teste e calculados os custos, usando os tempos medidos. Os resultados de cada bateria foi compilado usando média aritmética, e podem ser vistos na Tabela 4.1. A máquina número 1 foi dita ter custo de execução de R$ 0.0035 por segundo, e a máquina número 2 custo de R$ 0.0045 por segundo.

Tabela 4.1: Tabela Contendo os Resultados dos Testes de Performance.

		Máquina 1	Máquina 2	Total	Máximo
C99	tempo	00:14:08	00:01:04	00:15:12	00:14:08
	custo	R$ 2.96	R$ 0.28	R$ 3.25	R$ 2.96
ACO	tempo	00:03:35	00:22:34	00:26:09	00:22:34
	custo	R$ 0.75	R$ 6.09	R$ 6.84	R$ 6.09

Na Tabela 4.1 a linha C99 coresponde ao algoritmo proposto neste trabalho, enquanto a linha ACO coresponde ao escalonador ACOSched. Como pode ser visto o escalonador proposto teve uma performance melhor para o escalonamento do conjunto de tarefas testado, sendo de 30% a %60 melhor que o ACOSched em todos os parâmetros analisados. Essa diferença é atribuida principalmente ao fato que o ACOSched não leva em consideração o custo de execução.

4.4 Testes de Reatividade

O segundo tipo de teste foi pensado para demonstrar a reatividade do algoritmo para mudanças no tamanho das entradas. Para garantir a acurácia dos testes foi necessário montar um ambiente de testes separado da plataforma BioNimbuZ, isolando assim o escalonador e garantindo que qualquer impacto de performance seria oriundo do escalonador proposto.

4.4.1 Recursos

Para os testes de reatividade foi feita uma compilação de todas as 62 instâncias de nuvens dos provedores Amazon [2], Google [10] e Azure [6] combinados, para serem usadas como recursos. Esses dados foram colhidos em 28 de setembro de 2015 e estão descritos na Tabela 4.2, na qual são mostradas nas duas primeiras colunas o provedor e o tipo de instância; na terceira coluna o número de núcleos do processador; na quarta, a frequência do processador em GHz; na quinta coluna é apresentado o modelo do processador; na sexta a quantidade de memória em gigabytes, e por fim, nas últimas duas colunas a quantidade de espaço para armazenamento disponível e o custo da instância em dólares por hora. Dos 62 recursos pesquisados apenas 46 puderam ser utilizados, pois os recursos

não utilizados foram as instâncias da Azure porque não foi possível descobrir quais CPUs eram utilizadas.

Tabela 4.2: Instâncias Usadas nos Testes.

Provedora	**Instância**	**Cores**	**Clock**	**CPU**	**RAM**	**Storage**	**CustoCusto (US)**
Amazon	m4.large	2	2.5	E5-2676	8	-	0.126
Amazon	m4.xlarge	4	2.5	E5-2676	16	-	0.252
Amazon	m4.2xlarge	8	2.5	E5-2676	32	-	0.504
Amazon	m4.4xlarge	16	2.5	E5-2676	64	-	1.008
Amazon	m4.10xlarge	40	2.5	E5-2676	160	-	2.52
Amazon	m3.medium	1	2.6	E5-2670	3.75	4	0.067
Amazon	m3.large	2	2.6	E5-2670	7.5	32	0.133
Amazon	m3.xlarge	4	2.6	E5-2670	15	40	0.266
Amazon	m3.2xlarge	8	2.6	E5-2670	30	80	0.532
Amazon	c4.large	2	2.6	E5-2666	3.75	-	0.11
Amazon	c4.xlarge	4	2.6	E5-2666	7.5	-	0.22
Amazon	c4.2xlarge	8	2.6	E5-2666	15	-	0.441
Amazon	c4.4xlarge	16	2.6	E5-2666	30	-	0.882
Amazon	c4.8xlarge	36	2.6	E5-2666	60	-	1.763
Amazon	c3.large	2	2.8	E5-2680	3.75	16	0.105
Amazon	c3.xlarge	4	2.8	E5-2680	7.5	40	0.21
Amazon	c3.2xlarge	8	2.8	E5-2680	15	80	0.42
Amazon	c3.4xlarge	16	2.8	E5-2680	30	160	0.84
Amazon	c3.8xlarge	32	2.8	E5-2680	60	320	1.68
Amazon	r3.large	2	2.6	E5-2670	15	32	0.175
Amazon	r3.xlarge	4	2.6	E5-2670	30.5	80	0.35
Amazon	r3.2xlarge	8	2.6	E5-2670	61	160	0.7
Amazon	r3.4xlarge	16	2.6	E5-2670	122	320	1.4
Amazon	r3.8xlarge	32	2.6	E5-2670	244	320	2.8
Amazon	i2.xlarge	4	2.6	E5-2670	30.5	800	0.853
Amazon	i2.2xlarge	8	2.6	E5-2670	61	800	1.705
Amazon	i2.4xlarge	16	2.6	E5-2670	122	800	3.41
Amazon	i2.8xlarge	32	2.6	E5-2670	244	800	6.82
Amazon	d2.xlarge	4	2.5	E5-2676	30.5	2000	0.69
Amazon	d2.2xlarge	8	2.5	E5-2676	61	2000	1.38
Amazon	d2.4xlarge	16	2.5	E5-2676	122	2000	2.76
Amazon	d2.8xlarge	36	2.5	E5-2676	244	2000	5.52
Azure	A0	1	-	-	0.75	20	0.02
Azure	A1	1	-	-	1.75	70	0.06
Azure	A2	2	-	-	3.5	135	0.12
Azure	A3	4	-	-	7	285	0.24
Azure	A4	8	-	-	14	605	0.48
Azure	A5	2	-	-	14	135	0.25
Azure	A6	4	-	-	28	285	0.50
Azure	A7	8	-	-	56	605	1
Azure	A8	8	2.6	E5-2670	56	382	1.97
Azure	A9	16	2.6	E5-2670	112	382	4.47
Azure	A10	8	2.6	E5-2670	56	382	1.16

Continua na proxima pagina.

Tabela 4.2 – *Continuação da pagina anterior.*

Provedora	Instância	Cores	Clock	CPU	RAM	Storage	Custo (US)
Azure	A11	16	2.6	E5-2670	112	382	2.08
Azure	D1	1	-	-	3.5	50	0.094
Azure	D2	2	-	-	7	100	0.188
Azure	D3	4	-	-	14	200	0.376
Azure	D4	8	-	-	28	400	0.752
Azure	D11	2	-	-	14	100	0.238
Azure	D12	4	-	-	28	200	0.476
Azure	D13	8	-	-	56	400	0.857
Azure	D14	16	-	-	112	800	1.542
Google	n1-highmem-2	2	2.5	E5-?	13	-	0.096
Google	n1-highmem-4	4	2.5	E5-?	26	-	0.192
Google	n1-highmem-8	8	2.5	E5-?	52	-	0.384
Google	n1-highmem-16	16	2.5	E5-?	104	-	0.768
Google	n1-highmem-32	32	2.5	E5-?	208	-	1.536
Google	n1-highcpu-2	2	2.5	E5-?	1.80	-	0.058
Google	n1-highcpu-4	4	2.5	E5-?	3.60	-	0.116
Google	n1-highcpu-8	8	2.5	E5-?	7.20	-	0.232
Google	n1-highcpu-16	16	2.5	E5-?	14.40	-	0.464
Google	n1-highcpu-32	32	2.5	E5-?	28.80	-	0.928

Como um dos objetivos deste teste é descobrir a influência do número de recursos no tempo de execução do algoritmo proposto, foi montada uma segunda lista de recursos com apenas 13 recursos, os quais são vistos na Tabela 4.3. Este subconjunto de recursos é composto pelas menores instâncias de tipos distintos (por exemplo, instâncias Amazon m3, m4, c3, etc). Outra justificativa para essa lista reduzida é o fato de que os recursos do mesmo tipo (por exemplo m3, m4) possuem uma relação linear de crescimento de suas características. Por exemplo, se o custo por hora dobra, o número de *cores*, a RAM e, provavelmente, o *storage* também dobram. Dessa forma, dois tipos de testes foram gerados: testes com 46 recursos e testes com 13 recursos disponíveis.

Tabela 4.3: Conjunto Reduzido de Instâncias.

Provedora	Instância	Cores	Clock	CPU	RAM	Storage	Custo (US)
Amazon	m4.large	2	2.5	E5-2676	8	-	0.126
Amazon	m3.medium	1	2.6	E5-2670	3.75	4	0.067
Amazon	c4.large	2	2.6	E5-2666	3.75	-	0.11
Amazon	c3.large	2	2.8	E5-2680	3.75	16	0.105
Amazon	r3.large	2	2.6	E5-2670	15	32	0.175
Amazon	i2.xlarge	4	2.6	E5-2670	30.5	800	0.853
Amazon	d2.xlarge	4	2.5	E5-2676	30.5	2000	0.69
Azure	A8	8	2.6	E5-2670	56	382	1.97
Azure	A9	16	2.6	E5-2670	112	382	4.47
Azure	A10	8	2.6	E5-2670	56	382	1.16
Azure	A11	16	2.6	E5-2670	112	382	2.08

Continua na proxima pagina.

Tabela 4.3 – *Continuação da pagina anterior.*

Provedora	Instância	Cores	Clock	CPU	RAM	Storage	Custo (US)
Google	n1-highmem-2	2	2.5	E5-?	13	-	0.096
Google	n1-highcpu-2	2	2.5	E5-?	1.80	-	0.058

4.4.2 Tarefas

Para aumentar o número de testes realizados, e assim sendo possível chegar a conclusões mais significativas, foi usado um *log* de tarefas executadas em um *cluster*. O *log* utilizado foi o **LLNL Thunder**, proveniente da base *Parallel Workloads Archive* [12]. Esse *log* possui 128.662 tarefas no decorrer de cinco meses, independêntes entre si, e com custos de tempo entre 1 segundo e 100 horas.

Do *log* de tarefas usado para os testes, foram extraídos os tempos de execução e o tempo de início de cada tarefa. As 128.662 tarefas foram então divididas em *batches* de 1 a 32 tarefas. A divisão foi realizada baseando-se no tempo de entrada de cada tarefa e em uma janela de escalonamento. Assim, usando-se uma metodologia semelhante a usada no trabalho de Kessaci et al. [25], no qual foi usado o mesmo *log* de tarefas, e as mesmas foram escalonadas usando janelas de escalonamento crescentes, para aumentar o número de tarefas por *batch*. Nos testes realizados para o algoritmo proposto neste trabalho, foram usadas duas janelas de escalonamento, a primeira sendo de 50 segundos e a segunda de 500 segundos. A primeira janela produziu *batches* de no máximo 32 tarefas, e a segunda produziu *batches* de no máximo 72 tarefas. Vale ressaltar que o tempo de execução disponibilizado para o escalonador de 50 segundos para todos os testes. Dessa forma, quando combinou-se as classes de janelas de execução e o número de recursos disponíveis foram gerados quatro testes, os quais foram:

1. 13 recursos e 50 segundos de janela de escalonamento;
2. 13 recursos e 500 segundos de janela de escalonamento;
3. 46 recursos e 50 segundos de janela de escalonamento;
4. 46 recursos e 500 segundos de janela de escalonamento.

Assim sendo, o algoritmo testado, proposto neste trabalho, trata-se de um algoritmo de execução determinística, no sentido de que para várias execuções completas, com a mesma entrada, a saída será sempre a mesma. Dessa forma, sabendo-se que o tempo de execução sempre é limitado para janelas de execução de tamanho fixo, e assim, o tempo de execução não é analisado diretamente, a re-execução dos testes nos casos de *batches* que são executados completamente, não apresenta diferença de execuções anteriores. Para *batches* com execução incompleta, a única diferença encontrada foi a quantidade de nós de busca

visitados, mantendo os conjuntos de soluções intactos. Partindo desses entendimentos, os testes aqui realizados não foram executados mais de duas vezes, pois os resultados seriam idênticos.

4.4.3 Resultados dos Testes de Reatividade

Dos testes de reatividade redigidos neste trabalho, foi possível extrair quatro conclusões, relativas ao último *beam* e ao *beam* da última solução encontrada, conclusões sobre o algoritmo de poda e sobre o tamanho do conjunto de soluções. Assim, cada conclusão será descrita em duas partes, o significado do dado em questão e as próprias conclusões chegadas a partir desses dados.

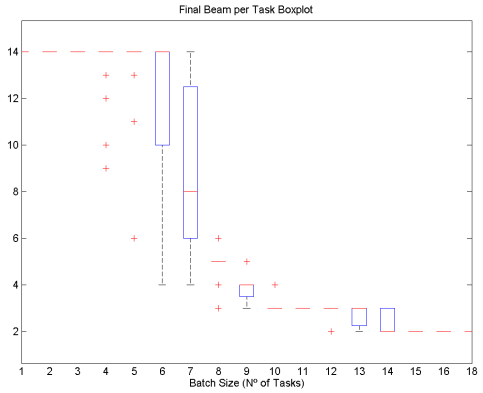
Dado que o domínio das variáveis de alguns gráficos eram demasiadamente grandes, e que nestes casos o resultado convergia para uma constante, vale ressaltar que os mesmos foram ajustados para uma melhor apresentação.

Beam Final

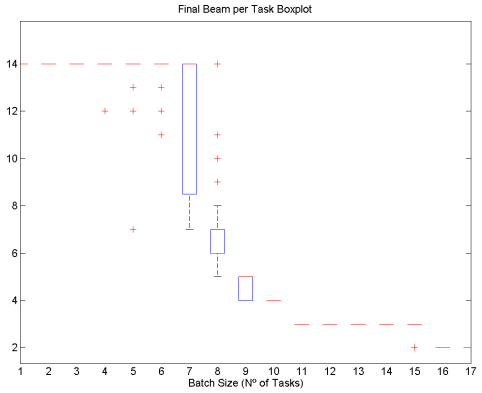
Por definição, ao se executar o *beam search* com o *beam* tendo o valor da largura máxima da árvore, este é equivalente à execução completa de uma busca combinacional. Pela implementação do algoritmo, dadas n tarefas, o *beam search* iterativo terminaria de executar assim que o *beam* assumisse o valor $n + 1$. Dessa forma, para 13 recursos, por exemplo, quando o valor de *beam* chegar a 14, a busca terá sido completa, no sentido de verificar todas as possíveis soluções.

Nas Figuras 4.2a, 4.2b, 4.2c e 4.2d tem-se no eixo x o tamanho do *batch* testado, e no eixo y o *beam* final quando os 50 segundos de escalonamento foram exauridos. Como visto nas figuras, o fator mais impactante no desempenho do algoritmo é o número de tarefas, que já era esperado dada a natureza NP-Completo do problema. Esta conclusão é vista quando se compara as Figuras 4.2a e 4.2b com as Figuras 4.2c e 4.2d, respectivamente, pois nota-se que o último *batch*, nas quais as medianas são 14 e 46 respectivamente, indicando a execução completa, é semelhante para ambas as comparações.

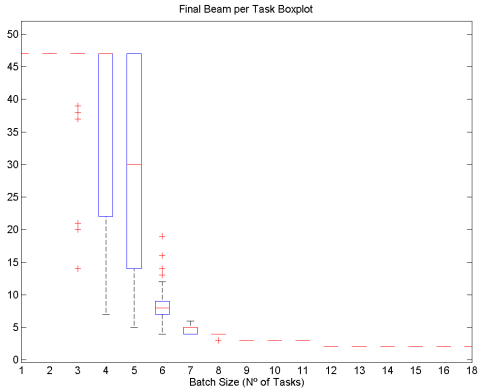
Assim, os dados apresentados nas Figuras 4.2a, 4.2b, 4.2c e 4.2d mostram que a execução completa do algoritmo para *batches* grandes é inviável, e que o impacto do número de recursos foi amortizado pelo algoritmo de poda, cujo inpacto será discutido em detalhes adiante.



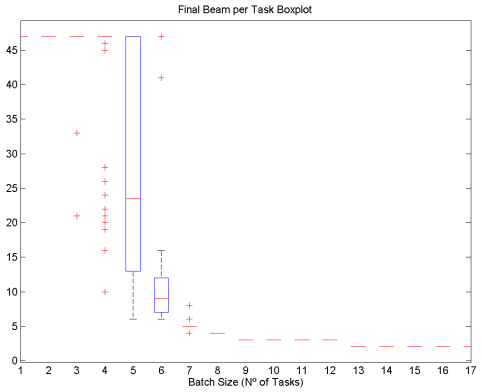
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



(c) *Batches* para 46 Recursos com 50 Segundos de Janela.



(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

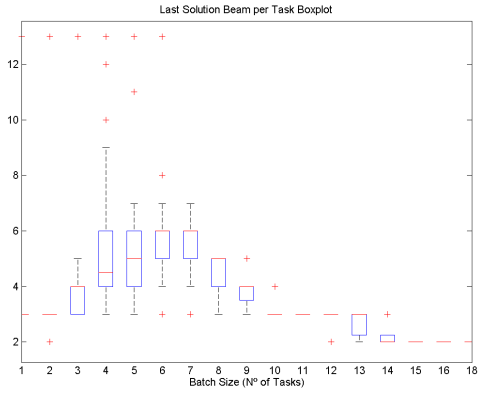
Figura 4.2: *Beam* Final pelo Número de Tarefas para Diferentes *Batches*.

Beam da Última Solução

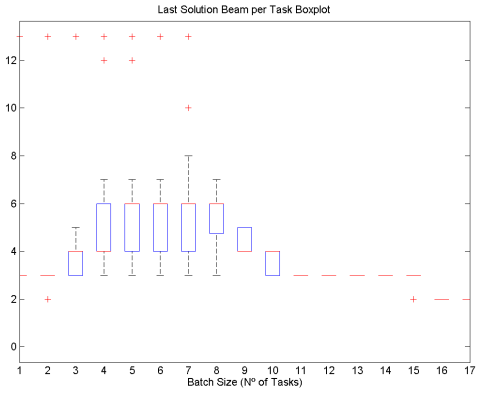
Outra métrica analisada a partir da execução do escalonador foi em qual *beam* ele estava quando encontrou a última solução que entrou para o conjunto de boas soluções. Esse experimento pode ser visto nas Figuras 4.3a, 4.3b, 4.3c e 4.3d onde o eixo y representa o *beam* na qual foi encontrada a ultima solução da busca e no eixo x tem-se o tamanho do *batch* testado. Dessa forma, como pode ser visto nas mesmas figuras, o algoritmo obtém a maior porcentagem de suas soluções (sendo até todas as soluções) nos primeiros *beams* da busca. Tal resultado é explicado pelo uso de pareto e do algoritmo de ordenação, sendo independente do tamanho da janela de escalonamento e do número de recursos disponibilizados.

Outra conclusão é que o algoritmo de ordenação é bom ao ponto de poder ser criada a heurística de buscar soluções apenas até o *beam* 4 ou 5, reduzindo assim o número de

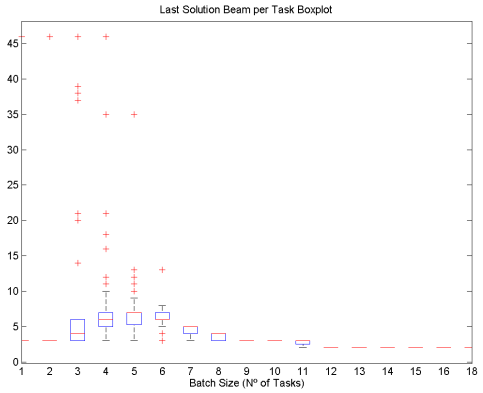
possibilidades a serem buscadas, e mantendo aproximadamente a mesma qualidade do conjunto de soluções. Esta conclusão é suportada pelas Figuras 4.2a, 4.2b, 4.2c e 4.2d, nas quais para os *batches* de tamanhos 1 a 5, foram, em sua maioria, buscas completas. Nessas buscas, por sua vez, as melhores soluções se encontram, também, em sua maioria (aproximadamente 75% do casos), com *beams* menores do que 6.



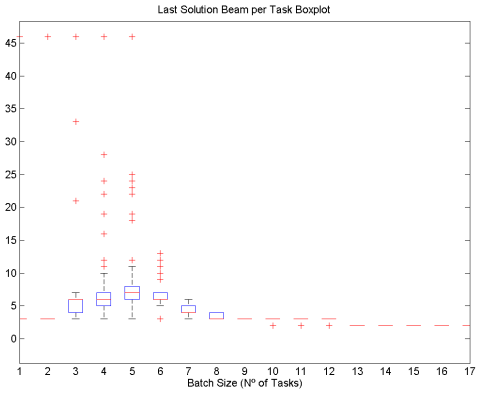
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



(c) *Batches* para 46 Recursos com 50 Segundos de Janela.



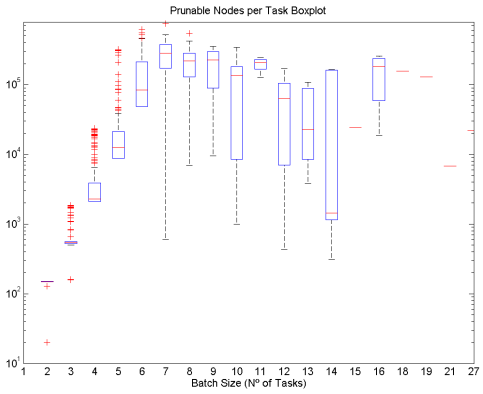
(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

Figura 4.3: *Beam* da Última Solução pelo Número de Tarefas para Diferentes *Batches*.

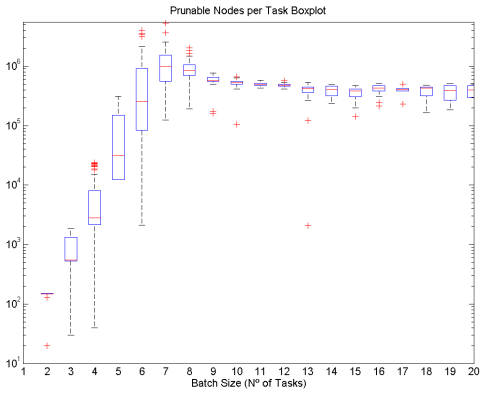
Poda de Nós de Busca

Em relação ao número de nós podáveis, a primeira observação que é feita é do comportamento logarítmico do mesmo, no qual esse número, embora tenha crescimento exponencial, possui um limite máximo ao qual a curva tende. Este comportamento, representado nas Figuras 4.4a, 4.4b, 4.4c e 4.4d, onde tem-se no eixo x o tamanho do *batch* testado e no

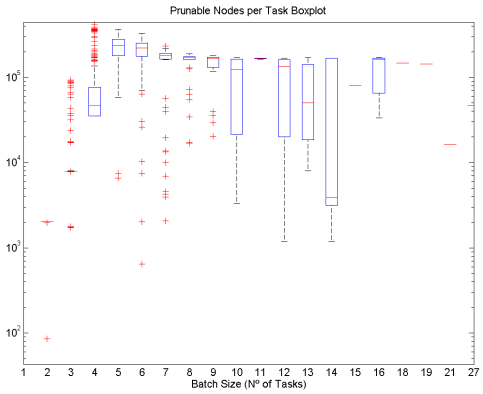
eixo y o número de nós podáveis em escala logarítmica, mostra apenas o limite da busca pelo tempo.



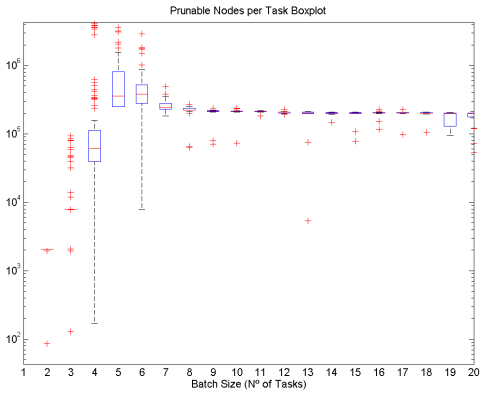
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



(c) *Batches* para 46 Recursos com 50 Segundos de Janela.

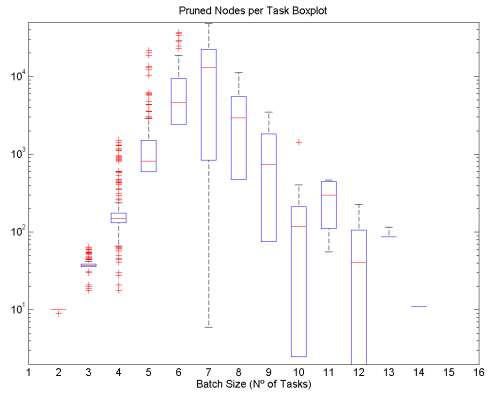


(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

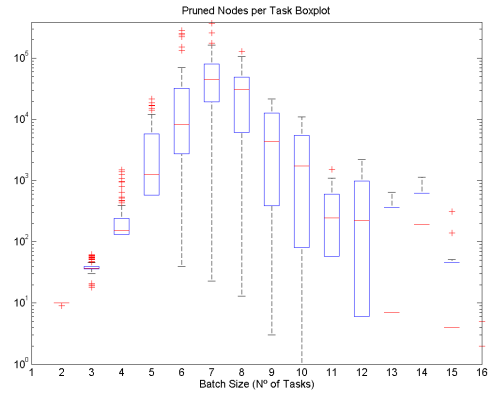
Figura 4.4: Número de Nós Podáveis pelo Número de Tarefas para Diferentes *Batches*.

Adicionalmente, ao comparar os gráficos de nós podáveis, como o da Figura 4.4a, com os gráficos dos nós podados, como o da Figura 4.5a, tendo como valor do eixo y o número de nós podados, é notado que um alto número de nós podáveis não necessariamente causa um alto número de nós podados. Esse fenômeno pode ser explicado pela natureza exponencial do número de nós quando variado o número de tarefas a serem escalonadas. Com o crescimento do número de nós, é natural que o número bruto de nós podáveis cresça, porém, sendo limitado pelo tempo máximo de execução. Tal relação não pode ser considerada verdadeira para o número de nós podados já que, dado que esse número é uma pequena fração do número total de nós podáveis e que o crescimento do número de nós é limitado pelo tempo. Dessa forma, é esperado que o número de nós podados reduza

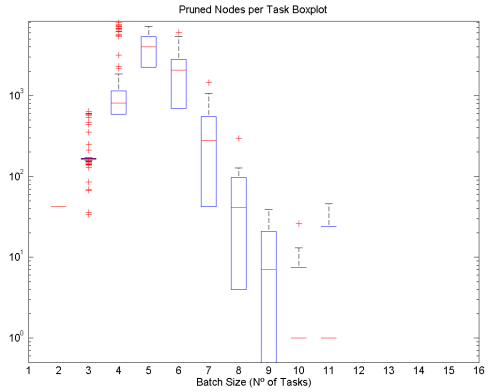
com o crescimento do número de tarefas, o que foi visto nas Figuras 4.5a, 4.5b, 4.5c e 4.5d.



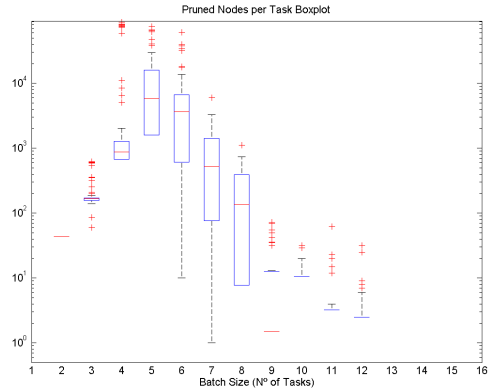
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



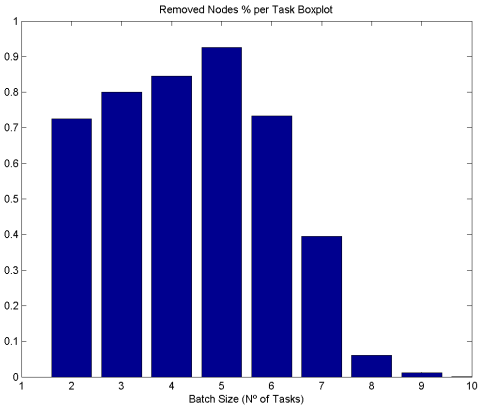
(c) *Batches* para 46 Recursos com 50 Segundos de Janela.



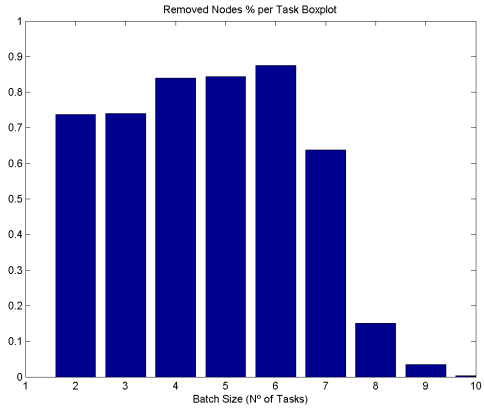
(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

Figura 4.5: Número de Nós Podados pelo Número de Tarefas para Diferentes *Batches*.

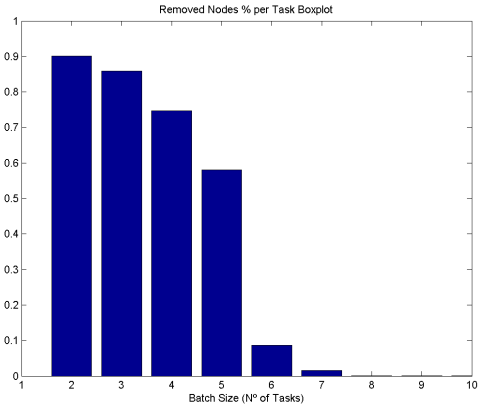
Finalmente, o foco do algoritmo de poda é remover o maior número possível de nós da busca, reduzindo assim o tempo de busca. Ao observar as Figuras 4.6a, 4.6b, 4.6c e 4.6d, tendo o eixo y representando a porcentagem média de nós removidos da busca para um conjunto de *batches* de mesmo tamanho, e tendo no eixo x o tamanho do *batch* testado, fica evidente que esse objetivo foi alcançado para *batches* com poucas tarefas, no qual a fração de nós removidos da árvore de busca chega em alguns casos a passar de 90% dos nós. Como mencionado no capítulo anterior, a ineficiência do algoritmo de poda é causada pela natureza do mesmo. Assim, para árvores mais profundas, o nó podado possui uma profundidade maior e, dessa forma, removendo proporcionalmente um número menor de nós quando comparado com nós podados mais próximos do nó raiz.



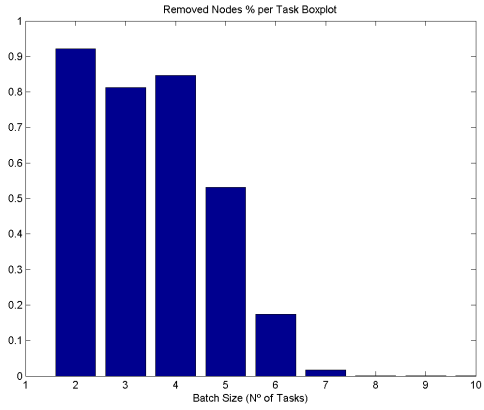
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



(c) *Batches* para 46 Recursos com 50 Segundos de Janela.

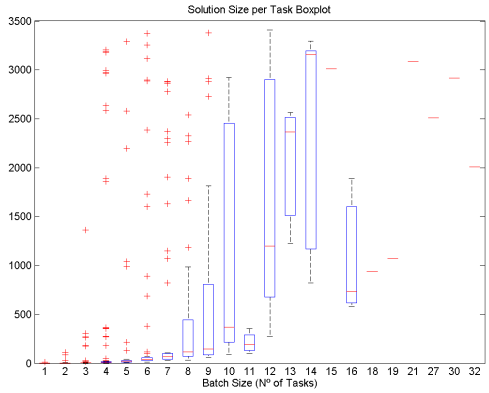


(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

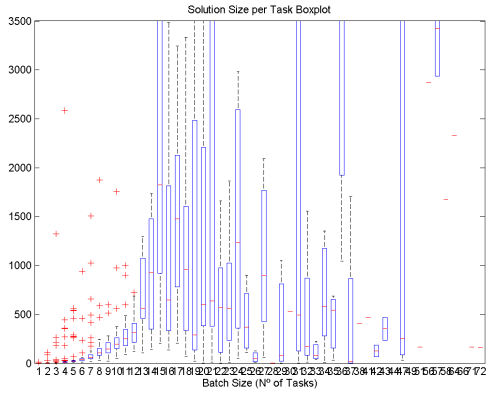
Figura 4.6: Número de Nós Removidos pelo Número de Tarefas para Diferentes *Batches*.

Tamanho dos Conjuntos de Soluções

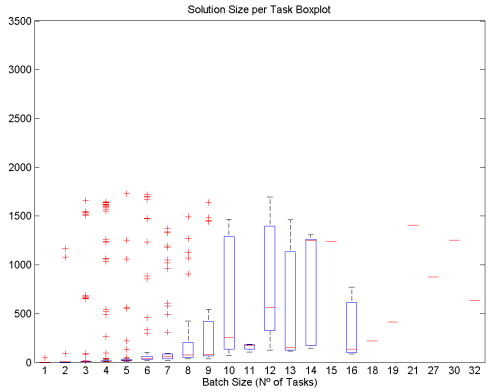
Assim, considerando os resultados do algoritmo de ordenação, é possível afirmar que a maioria das melhores soluções globais é encontrada nas primeiras etapas do algoritmo. Dessa maneira, existe um alto número de soluções, independente do tamanho dos *batches* e do número de recursos disponíveis. Essa conclusão é suportada pelos gráficos das Figuras 4.7a, 4.7b, 4.7c e 4.7d, nos quais o eixo y corresponde ao número de soluções encontradas para cada grupo de *batches*, tendo como limite máximo de visualização, 15 soluções.



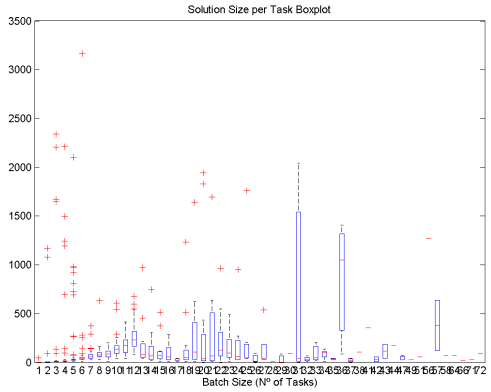
(a) *Batches* para 13 Recursos com 50 Segundos de Janela.



(b) *Batches* para 13 Recursos com 500 Segundos de Janela.



(c) *Batches* para 46 Recursos com 50 Segundos de Janela.



(d) *Batches* para 46 Recursos com 500 Segundos de Janela.

Figura 4.7: Tamanho do Conjunto de Solução pelo Número de Tarefas para Diferentes *Batches*.

4.4.4 Progressão do Conjunto de Soluções

O algoritmo proposto melhora o conjunto de soluções no decorrer da busca. Entretanto é interessante ver um exemplo desta progressão nos diversos estágios de busca. Para tal, foi extraido uma execução exemplo com 6 tarefas a serem escalonadas e 13 recursos disponiveis. Esta execução foi dividida nos estágios 1, 2, 3 com *beam* de tamanho 2, 3 com *beam* de tamanho 3 e 3 com *beam* de tamanho 4. Estes resultados encontram-se nas Figuras 4.8, 4.9, 4.10 e 4.11.

Para cada etapa da execução foram mostradas as soluções encontradas no formato de uma frente de pareto, sendo as soluções identificadas pelo estágio que foram encontradas e se ainda pertencem ao conjunto de soluções ótimas.

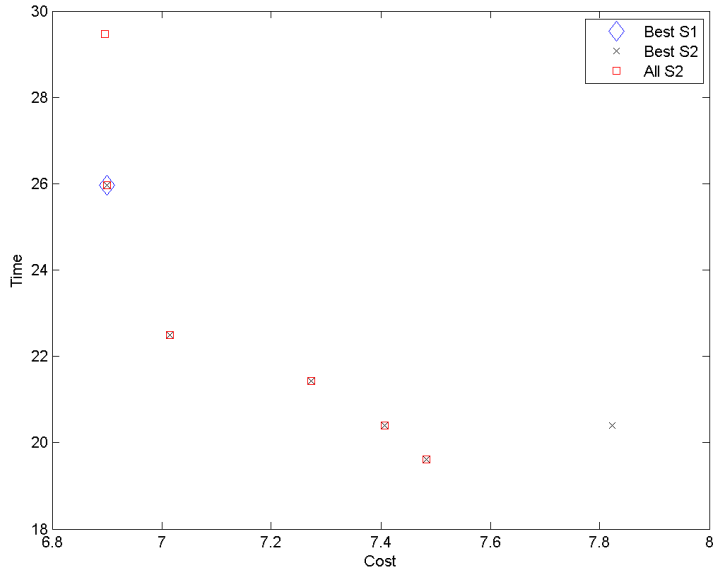


Figura 4.8: Curva de Pareto ao Final do Estágio 2.

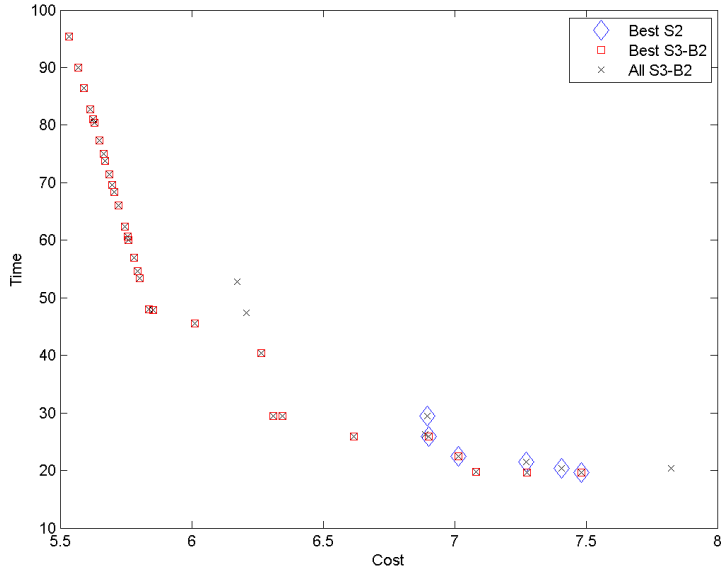


Figura 4.9: Curva de Pareto ao Final do Estágio 3 com *Beam* 2.

As soluções em todos os graficos são divididos em três grupos, os quais são grupo de soluções ótimas encontradas em um primeiro estágio, grupo de soluções ótimas encontradas no segundo estágio, e grupo de todas as soluções encontradas que são ótimas ou já foram ótimas e agora são estritamente domidadas por outra solução (i.e. não-ótimas).

Na Figura 4.8 tem-se o resultado do primeiro estágio (busca gulosa) que retorna apenas uma solução, comparado com o segundo estágio (*limited discrepancy*). Como é visto na Figura 4.8, a solução gulosa conseguiu retornar um resultado que se manteve no conjunto ótimo após o segundo estágio. Também foi visto que o conjunto de soluções novas, encontradas pelo segundo estágio foi de 6 soluções, como previsto como sendo igual ao número de tarefas.

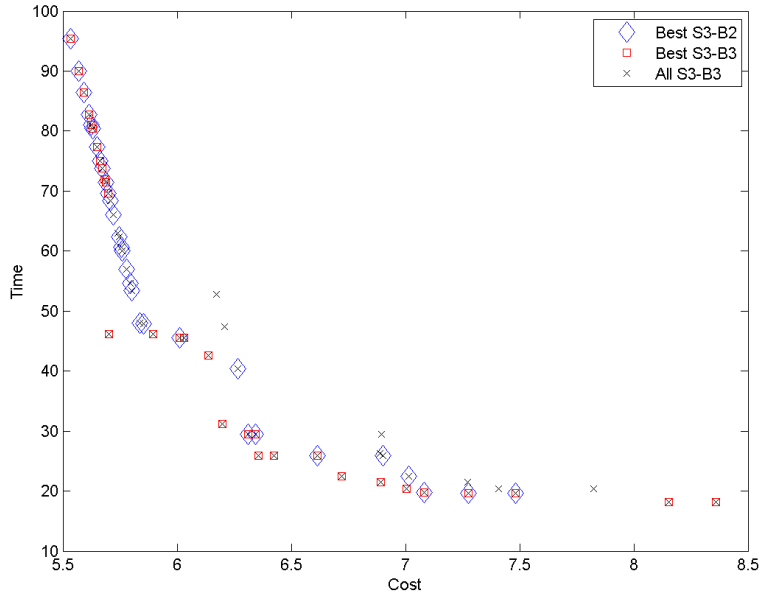


Figura 4.10: Curva de Pareto ao Final do Estágio 3 com *Beam* 3.

Com o início do *beam search* foram encontradas mais 30 soluções, sendo duas já removidas do conjunto ótimo no mesmo estágio, o que pode ser visto na Figura 4.9. Das soluções encontradas no estágio anterior apenas 3 soluções se mantiveram. Isto mostra que embora o conjunto de soluções do segundo estágio tenha sido sub-ótimo, quando comparado com este estágio, a ordem dos nós de busca não é ótima. Presume-se que isto pode ser melhorado com a ordenação das tarefas.

Finalmente, nas figura 4.10 e 4.11 a continuação do processo de busca iterativa com o *beam search* é recompensado com um cresscente conjunto de soluções ótimas.

Este exemplo foi escolhido por mostrar um alto número de tarefas em todos os estágios descritos. Também é valido destacar que não houveram mais soluções encontradas após o

beam 4 do estágio três, e que este caso foi executado até a sua completude (todos os nós, com exceção dos nós podados, foram buscados).

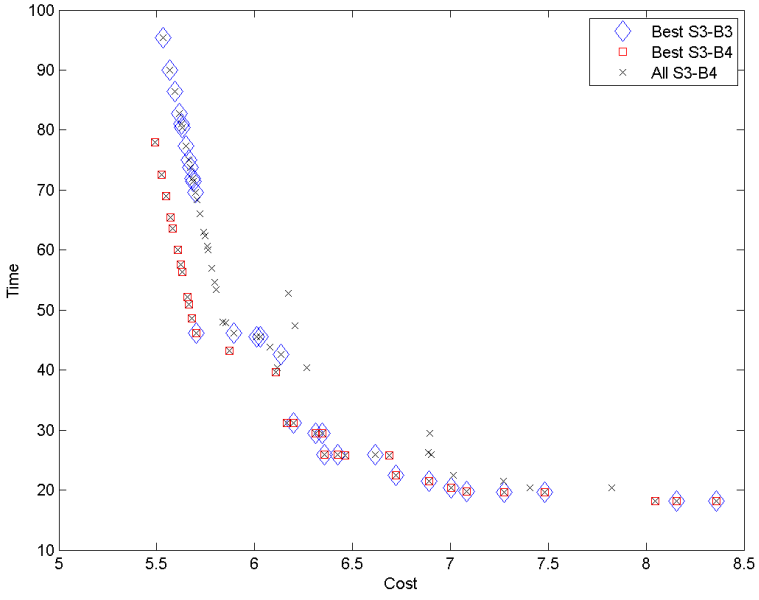


Figura 4.11: Curva de Pareto ao Final do Estágio 3 com *Beam* 4.

4.5 Considerações Finais

Este capítulo serviu como uma complementação do capítulo anterior, demonstrando empiricamente os pontos positivos e negativos do algoritmo de escalonamento de tarefas proposto. Como foi visto, o algoritmo completo teve uma baixa sensibilidade a variações no número de recursos disponibilizados para escalonamento devida a estratégia de poda.

Além disso, foi possível concluir que o algoritmo de ordenação, que usa pareto, teve uma boa performance individualmente, dado que a última solução encontrada, na maioria dos casos, foi encontrada nos primeiros *beams* da busca, e também quando comparado com o algoritmo ACOSched. Finalmente, vendo que os casos onde essas boas soluções, que foram encontradas no começo da busca, foram casos de busca completa (i.e. todos os nós de busca foram visitados ou podados), chega-se a conclusão que pode ser feita a busca incompleta (por exemplo, com a busca chegando até o *beam* 5 ou 6) e a qualidade da solução terá sido pouco impactada.

Existe, porém, a limitação do número de tarefas a serem escalonadas, onde o algoritmo torna-se ineficiente para *batches* com mais de 7 tarefas, que não foi resolvido neste trabalho. O próximo capítulo descreverá possíveis soluções para esse problema, e outros aperfeiçoamentos que melhorariam a performance do algoritmo proposto.

Capítulo 5

Conclusão

Este trabalho foi feito com o propósito de propor uma nova política de escalonamento para a plataforma BioNimbuZ. Essa política foi implementada com o objetivo de considerar ambas as características de tempo de execução e custo. Para tal, foi implementado um algoritmo combinacional determinístico e completo no sentido de optimalidade, usando pareto para a representação da dualidade dos objetivos custo e tempo de execução.

Nos testes redigidos foi provada a completude do algoritmo, mesmo com o uso de uma estratégia de poda original, feita para essa classe de algoritmos. Além disso, para uma execução incompleta (execução parada prematuramente) foi demonstrado também que o algoritmo é *any-time* e incremental, tendo uma boa estratégia de ordenação dos nós a serem buscados.

Outro ponto relevante é sobre o tratamento de *pipelines* enviados para o BioNimbuZ contendo tarefas, as quais possam ter dependências. O BioNimbuZ ainda não possui uma política de escalonamento que leve em consideração tais dependências para otimizar o processo de escalonamento. A implementação de tal política seria demasiadamente complexa, já que para tal seria necessário o conhecimento prévio do tempo de execução de cada tarefa, e qualquer erro no processo de cálculo do custo de tempo impactaria severamente na eficiência do algoritmo. Contudo, é importante destacar que a abordagem descrita neste trabalho não foi gerada para resolver este problema.

Também foi visto que o algoritmo aqui proposto mantém a complexidade NP-Dificil do problema de escalonamento, sendo assim, extremamente sensivel ao número de tarefas a serem escalonadas. Além disso, o equacionamento de custo e de tempo de execução ainda é simples, levando em consideração apenas o custo de execução de uma tarefa e a capacidade de processamento de um recurso.

Para os problemas descritos anteriormente, existem maneiras de resolvé-los com baixo impacto no que já foi implementado. Uma maneira de mitigar o impacto do número de tarefas é a aglomeração de conjuntos de tarefas com dependências exclusivamente internas.

Por exemplo, tem-se três tarefas T1, T2 e T3, de forma que T3 deve depender apenas de T2 e T2 depende apenas de T1. Assim, T1, T2 e T3 podem ser aglomeradas em apenas uma tarefa. Outra melhoria para a busca é a remoção de isomorfismos, ou seja, remoção de nós de busca que teriam o mesmo resultado que outro nó já buscado.

Outra otimização possivel é a realização da etapa de *beam search* apenas para os n primeiros nós, n sendo um valor pequeno (isto é, de 4 a 7 nós). Isso é possivel de ser feito devido aos resutados do algoritmo de ordenação de busca, que mostrou que para execuções completas, todas as soluções, na maior parte das vezes, foram encontradas no começo da busca. Essa otimização, entretanto, removeria a completude do algoritmo.

Todas as otimizações supramencionadas não diminuem a complexidade do algoritmo. Assim, o problema de escalonamento para grandes números de tarefas é apenas mitigado, e não resolvido. Uma forma para resolver esse problema é a execução do algoritmo de escalonamento de maneira iterativa em subconjuntos de tarefas. Por outra forma, as tarefas seriam divididas em subconjuntos menores, que podem ser executados em tempo hábil. Para cada solução do conjunto de soluções da iteração anterior, será executado então o algoritmo novamente. Para tal, é necessária a ordenação da lista de tarefas já que tal ordem definirá quais tarefas terão maior prioridade de alocação.

Finalmente, para melhorar a acurácia dos resultados do escalonamento para o domínio do problema, é interessante o refinamento das equações de custo e de tempo de execução, adicionando características de sistemas em nuvens como custo de *dispatch* de tarefas, localidade e latência de arquivos, e taxa de transmissão de rede.

Referências

[1] *Encyclopedia of Parallel Computing*. 292-295, Springer. 1

[2] Amazon elastic computing 2. aws.amazon.com, acessado em dezembro de 2015. 2, 14, 28

[3] Apache. www.apache.org, acessado em dezembro de 2015. 6

[4] Apache avro. avro.apache.org, acessado em dezembro de 2015. 6

[5] Apache zookeeper. ZooKeeper.apache.org, acessado em dezembro de 2015. 6

[6] Azure cloud computing. azure.microsoft.com/en-us/, acessado em dezembro de 2015. 14, 28

[7] Bowtie, an ultrafast memory-efficient short read aligner. bowtie-bio.sourceforge.net, acessado em dezembro de 2015. 27

[8] Cdc historical timeline. www.cbi.umn.edu/collections/cdc/histtimeline.html, acessado em dezembro de 2015. 1

[9] Eucalyptus. www8.hp.com/us/en/cloud/helion-eucalyptus-overview.html, acessado em dezembro de 2015. 6

[10] Google cloud plataform. cloud.google.com, acessado em dezembro de 2015. 14, 28

[11] Opennebula. opennebula.org/, acessado em dezembro de 2015. 6

[12] Parallel workloads archive. www.cs.huji.ac.il/labs/parallel/workload/, acessado em dezembro de 2015. 31

[13] Salesforce. www.salesforce.com/company/, acessado em dezembro de 2015. 2

[14] Xen hypervisor project. www.xenproject.org/, acessado em dezembro de 2015. 27

[15] M. Armbrust, O. Fox, R. Griffith, A. D. Joseph, Y. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: a berkeley view of cloud computing. *Magazine Communications of the ACM*, 53(4):50–58, 2010. 4

[16] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, e A. Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010. 4

[17] Heitor Henrique de Paula Moraes Costa. Controle de acesso na plataforma de nuvem federada bionimbuz. 2015. 8

[18] European Telecommunications Standards Institute (ETSI). Cloud standards coordination final report. *ESTI*. 11 de dezembro de 2013. acessado em dezembro de 2013. http://csc.etsi.org/. 4, 5

[19] P. Eugster, W. Culhane, C. Jayalath, K. Kogan, e J. Stephen. Cloud federation and distribution. *Encyclopedia of Cloud Computing. Wiley-IEEE*, pages 1–16. 5, 6

[20] M. Fazio, A. Celesti, M. Villari, e A. Puliafito. How to enhance cloud architectures to enable cross-federation: Towards interoperable storage providers. *IEEE International Conference on Cloud Engineering*, pages 480–486, 2015. 5

[21] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 8

[22] Simson L Garfinkel. Architects of the information society: Thirty-five years of the laboratory for computer science at mit. *MIT Press*, 1999. 1

[23] R. L. Graham, E. L. Lawler, J. K. Lenstra, e A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979. 10

[24] W. D. Harvey e M. L. Ginsberg. Limited discrepancy search. *IJCAI*, pages 607–615, 1995. 16, 20

[25] Y. Kessaci, N. Melab, e E.-Ghazali Talbi. A pareto-based metaheuristic for scheduling hpc applications on a geographically distributed cloud federation. *Cluster Computing*, 16(3):451–468, 2013. 9, 11, 31

[26] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, e M. Kunze. Cloud federation. *Proceedings of the 2nd International Conference on Cloud Computing and GRIDs and Virtualization*, pages 32–38, 2011. 5

[27] J. C. R. Licklider e R. Taylor. The computer as a communication device. *Science and Technology*, 76:21–31, 1968. 1

[28] J. Liu, X. Luo, X. Zhang, F. Zhang, e B. Li. Job scheduling model for cloud computing based on multi- objective genetic algorithm. *IJCSI International Journal of Computer Science Issues*, 10(3):134–139, 2013. 9, 11

[29] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, e I. M. Llorente. Scheduling strategies for optimal service deployment across multiple clouds. *Future Generation Computer Systems*, 29(6):1431–1441, 2013. 9, 11, 12

[30] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, e I. M. Llorente. Cost optimization of virtual infrastructures in dynamic multi-cloud scenarios: Cost optimization of virtual infrastructures in dynamic multi-cloud scenarios. *Concurrency and Computation: Practice and Experience*, 27(9):2260–2277, 2015. 9, 11, 12

[31] C. A. Mattson e A. Messac. Pareto frontier based concept selection under uncertainty and with visualization. *Optimization and Engineering*, 6(1):85–115, 2005. 15

[32] P. Mell e T. Grance. (2011). the nist definition of cloud computing. (NIST Special Publication 800-145), Gaithersburg: U.S. Department of Commerce. 4, 5

[33] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs, Third Edition*. Springer-Verlag, New York, 1999. 9

[34] Breno R. Moura, Deric L. Bacelar, Edward Ribeiro, Aletéia P. F. Araújo, Maristela T. Holanda, e Maria E. M. T. Walter. Política de armazenamento em uma infraestrutura de nuvens federadas para aplicações de bioinformática. *XIV Simpósio em Sistemas Computacionais*, 2013. 8

[35] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999. 11, 12

[36] Christos H. Papadimitriou e Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, 2013. 26

[37] I. Qureshi. Cpu scheduling algorithms: A survey. *Int. J. Advanced Networking and Applications*, 5(4):1968–1973, 2014. 9, 14

[38] S. M. Rubin e R. Reddy. The locus model of search and its use in image interpretation. *IJCAI*, 2:590, 1977. 13, 14, 20, 21

[39] B. Sareni e L. Krähenbühl. Fitness sharing and niching methods revisited. *Evolutionary Computation and IEEE Transactions*, 2(3):97–1998, 1998. 10

[40] J. L. L. Simarro, R. Moreno-Vozmediano, R. S. Montero, e I. M. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. *High Performance Computing and Simulation (HPCS)*, pages 1–7, 2011. 9, 11, 12

[41] S. Sotiriadis, N. Bessis, e N. Antonopoulos. Towards inter-cloud schedulers: A survey of meta-scheduling approaches. pages 59–66, 2011. 10

[42] AcoSched: um escalonador para o ambiente de Nuvem Federada ZooNimbus por G. S. S. de Oliveira Universidade de Brasília. 2013, 66 páginas. 8, 9, 27

[43] Saldanha H. V. Bionimbus: uma arquitetura de federação de nuvens computacionais híbrida para a execução de workfows de bioinformática. *Master's thesis and Departamento de Ciência da Computação and Universidade de Brasília*, 2012. 6

[44] D. A. Van Veldhuizen e G. B. Lamont. Evolutionary computation and convergence to a pareto front. *Late breaking papers at the genetic programming 1998 conference*, pages 221–228, 1998. 11, 15

[45] Lizhe Wang e Gregor von Laszewski. Scientific cloud computing: Early definition and experience. *10th IEEE International Conference on High Performance Computing and Communications.*, pages 825–830, 2008. 1

[46] J. Yu e R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. *Workflows in Support of Large-Scale Science*, pages 1–10, 2006. 9, 11

[47] W. Zhang. Complete anytime beam search. *Association for the Advancement of Artificial Intelligence (AAAI-98)*, 1998. 21, 23

[48] R. Zhou e E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. *ICAPS*, pages 90–98, 2005. 16